

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN

ESCUELA PROFESIONAL DE CIENCIA DE LA
COMPUTACIÓN



ALGORITMO PARALELOS

Práctica de 4to Laboratorio

Alumno:

Eddy René Cáceres
Huacarpuma

CUI:

20101934

24 de abril de 2017

Índice general

1. Introducción	2
1.1. Implementar el problema Matriz-Vector usando PTthreads	2
1.2. Implementar el problema del Calculo de PI	4
1.3. Realizar pruebas y cambios de la tabla 4.1 del libro (Busy Waiting y Mutex)	6
1.4. Implementar la lista enlazada multithreading y replicar las tablas 4.3 y 4.4	9
1.5. Realizar experimentos y replicar el cuadro 4.5	9
1.6. Implementar el problema presentado en la sección 4.11 del uso de strtok.	10

Capítulo 1

Introducción

En el presente trabajo se realizó la implementación de los ejemplos presentados en el capítulo 4 del Libro "Introduction to parallel programming" titulado "Shared-Memory Programming with Pthreads". El repositorio de los programas es :

[https : //github.com/eddyrene/Laboratorio/tree/master/4to%20Laboratorio](https://github.com/eddyrene/Laboratorio/tree/master/4to%20Laboratorio)

- Multiplicación Vector-Matriz : mm.cpp
- Calculo de pi , standar, busy waiting y mutex : pi.cpp
- Tokenización de textos: tokenize.cpp

1.1. Implementar el problema Matriz-Vector usando PThreads

```
void *Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i;
    int j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
```

```
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }

    return NULL;
}

int main(int argc, char* argv[]) {

    m = 8;
    n = 8000000;
    x = new double[n];
    y = new double[m];
    A = new double[m*n];
    struct timespec start, finish;
    double elapsed;

    long thread;
    pthread_t* thread_handles;

    thread_count = strtol(argv[1], NULL, 10);
    //cout<<"numero de thread: "<< thread_count<<endl;
    thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));

    //cout<<"vector "<<endl;
    rand_vector(x, n);
    //print_vector(x, n);

    //cout<<"matriz"<<endl;
    rand_matrix(A, m, n);
    //print_matrix(A, n, m);

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Pth_mat_vect, (void*) thread);

    //printf("Hello from the main thread\n");
```

```
for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);

clock_gettime(CLOCK_MONOTONIC, &finish);
//cout<<"resultado"<<endl;
//print_vector(y,m);
elapsed = (finish.tv_sec - start.tv_sec);
elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
cout<<"tiempo de ejecucion "<<elapsed<<endl;

free(thread_handles);
delete [] A;
delete [] x;
delete [] y;
return 0;

}
```

En mpi se tenia que trabajar de mejor manera los datos, con las funciones **MPI_scatter** y **MPI_allgather**. En Pthreads hay una mayor sencillas , los threads se dividen sobre el vector usando como matriz para realizar la multiplicación.

1.2. Implementar el problema del Calculo de PI

Es sabido que el paralelismo es un herramienta para el calculo rápido , sin embargo es importante tener en cuenta que pueden existir partes criticas en el programa, veremos esto con un programa de calculo de PI.

[H]

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;

    long long i;
    cout<<"n: "<<n<<endl;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;
```

```

if (my_first_i % 2 == 0) /* my first i is even */
    factor = 1.0;
else /* my first i is odd */
    factor =- 1.0;
cout<<"se estas "<<endl;
for (i = my_first_i; i < my_last_i; i++, factor =- factor) {
    sum += factor/(2*i+1);
}
return NULL;
}

```

```

mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Labortorio $ ./p_normal 2
n: 1000000000
se estas
n: 1000000000
se estas
resultado : 3.14079
tiempo de ejecucion 1.71784
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Labortorio $ ./p_normal 3
n: 1000000000
se estas
n: 1000000000
se estas
resultado : 3.13937
tiempo de ejecucion 2.53011
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Labortorio $ ./p_normal 4
n: 1000000000
se estas
n: 1000000000
se estas
resultado : 3.15007
tiempo de ejecucion 2.53011
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Labortorio $ ./p_normal 5
n: 1000000000
se estas
n: 1000000000
se estas
resultado : -0.000357743
tiempo de ejecucion 2.38778

```

Variable	Valor
1	1000000000
2	1000000000
3	1000000000
4	1000000000
5	1000000000
6	1000000000
7	1000000000
8	1000000000
9	1000000000
10	1000000000
11	1000000000
12	1000000000
13	1000000000
14	1000000000
15	1000000000
16	1000000000
17	1000000000
18	1000000000
19	1000000000
20	1000000000
21	1000000000
22	1000000000
23	1000000000
24	1000000000
25	1000000000
26	1000000000
27	1000000000
28	1000000000
29	1000000000
30	1000000000
31	1000000000
32	1000000000
33	1000000000
34	1000000000
35	1000000000
36	1000000000
37	1000000000
38	1000000000
39	1000000000
40	1000000000

Figura 1.1: Ejecución con diferentes threads del calculo de Pi, usando semáforos

PI	Valor
1	3,14159
2	3,14079
3	3,13937
4	315.007
5	2,38778

Figura 1.2: Cálculo de Pi con diferentes threads

Como se aprecia en la figura, cuando se ejecuta con una hebra se obtiene el valor correcto sin embargo mientras se incrementa las hebras el valor se va alejando.

1.3. Realizar pruebas y cambios de la tabla 4.1 del libro (Busy Waiting y Mutex)

Lo que sucede en el programa anterior que los threads estan accediendo a la misma variable global (este es uno de los precios) , y la modifican. Para e evitar este problema una tecnica es busy-waiting que detiene el programa hasta que solo el thread elegido está listo.

```
[H]
void* Thread_sum_bs(void* rank) {
    long my_rank = (long) rank;
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
        my_sum += factor/(2*i+1);

    while (flag != my_rank);
    sum += my_sum;
```

```

    if (my_rank < thread_count-1)
        flag++;

    return NULL;
}

```

```

mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./pi_bw 1
resultado: 3.14159
tiempo de ejecucion 1.15328
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./pi_bw 2
resultado : 3.14159
tiempo de ejecucion 1.20847
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./pi_bw 4
resultado : 3.14159
tiempo de ejecucion 1.22072
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./pi_bw 8
resultado : 3.14159
tiempo de ejecucion 1.48619
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./pi_bw 16
resultado : 3.14159
tiempo de ejecucion 1.68212
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./pi_bw 32
resultado : 3.14159
tiempo de ejecucion 2.10529

```

Figura 1.3: Captura de pantalla uso de busy-waiting

Sin embargo busy-waiting generalmente no es la solución ideal para limitar el acceso a secciones críticas. Mutex (mutual exclusion). puede ser usado para un thread excluya a los demás

```

void* Thread_sum_mutex(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;
    double my_sum = 0.0;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
}

```



```

for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
    my_sum += factor/(2*i+1);
}
pthread_mutex_lock(&mutex);

sum += my_sum;
pthread_mutex_unlock(&mutex);

return NULL;
}

```

Threa	Busy-Wait	Mutex
1	1,15	1,15
2	1,2	1,17
4	1,22	1,24
8	1,48	1,26
16	1,68	1,18
32	2,1	1,26

Figura 1.4: Captura de pantalla, uso de mutex

Como se aprecia en el código de arriba se señala con **lock** y **unlock** para la parte crítica.

A continuacion se muestra una tabla de comparativa de los 2 tecnicas.

Threa	Busy-Wait	Mutex
1	1,15	1,15
2	1,2	1,17
4	1,22	1,24
8	1,48	1,26
16	1,68	1,18
32	2,1	1,26

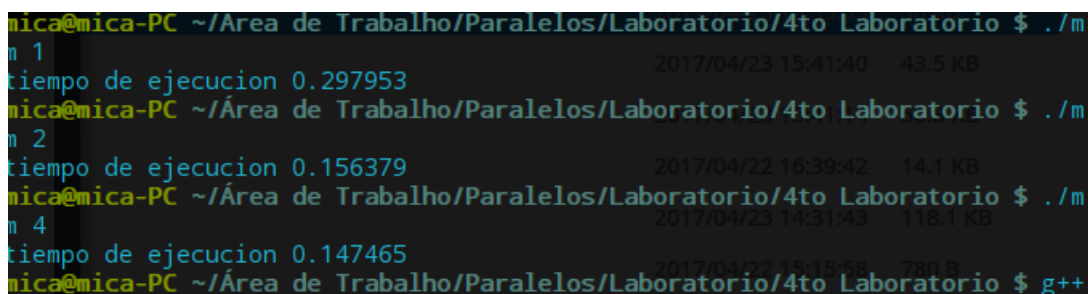
Figura 1.5: Comparacion entre busy-waiting y mutex

1.4. Implementar la lista enlazada multithreading y replicar las tablas 4.3 y 4.4

1.5. Realizar experimentos y replicar el cuadro 4.5

En la tabla original 4.5 del libro se hace la comparación del tiempo de ejecución de la multiplicación vector-matriz, en esta sección se estudia la memoria caché. Se incluye además una columna de eficiencia:

$$Eficiencia = \frac{Tiempo de ejecución serial}{(Número de hilos) \times tiempo de ejecución paralela}$$

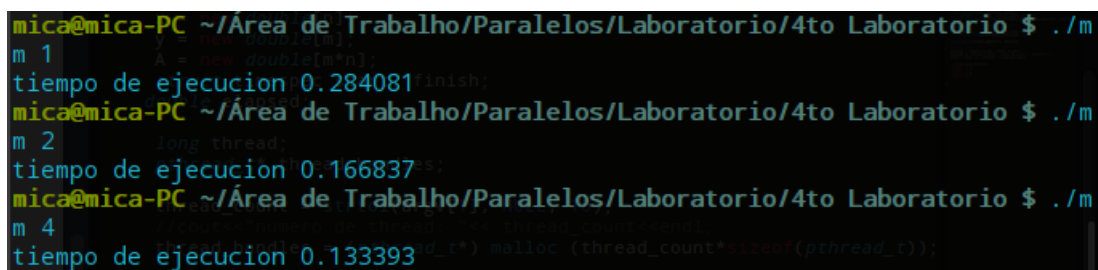


```

mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 1
tiempo de ejecución 0.297953
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 2
tiempo de ejecución 0.156379
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 4
tiempo de ejecución 0.147465
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ g++

```

Figura 1.6: Captura de pantalla de ejecución 8,000,000 x 8



```

mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 1
tiempo de ejecución 0.284081
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 2
tiempo de ejecución 0.166837
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 4
tiempo de ejecución 0.133393

```

Figura 1.7: Captura de pantalla de ejecución 8,000 x 8,000

```
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 1
tiempo de ejecucion 0.271316
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 2
tiempo de ejecucion 0.146968
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $ ./m
m 4
tiempo de ejecucion 0.185293
mica@mica-PC ~/Área de Trabalho/Paralelos/Laboratorio/4to Laboratorio $
```

Figura 1.8: Captura de pantalla de ejecucion 8 x 8,000,000

	Dimension de matriz					
	8,000,000 x 8		8000 x 8000		8 x 8,000,000	
Threads	Tiempo	Efic	Tiempo	Efic	Tiempo	Efic
1	0,2979	1	0,284	1	0,2713	1
2	0,1563	0,952975048	0,1668	0,8513189448	0,1469	0,9234172907
4	0,1474	0,2650949796	0,1333	0,5326331583	0,1852	0,3662257019

Figura 1.9: Tabla de comparacion entre multiplicacion vector-matriz

1.6. Implementar el problema presentado en la sección 4.11 del uso de strtok.

En esta sección se revisa la seguridad del codigo al ser ejecutada simultaneamente, sin causar problemas.

Para ejemplificar eso se realiza la funcion tokenizar, se utiliza **strtok** de **string**, sin embargo cuando se realizan los experimentos esto no son satisfactorios y la razon es porque esta funcion no asegura que se comportará bien al ser ejecutada por varios threads. Se sugiere entonces la funcion: **strtok_r**, la diferencia entre estas funciones es que la segunda utiliza un puntero para marcar hasta que lugar se recorrió en la cadena a tokenizar.

```
void *Tokenize(void* rank) {
    long my_rank = (long) rank;
    int count;
    int next = (my_rank + 1) % thread_count;
    char *fg_rv;
```

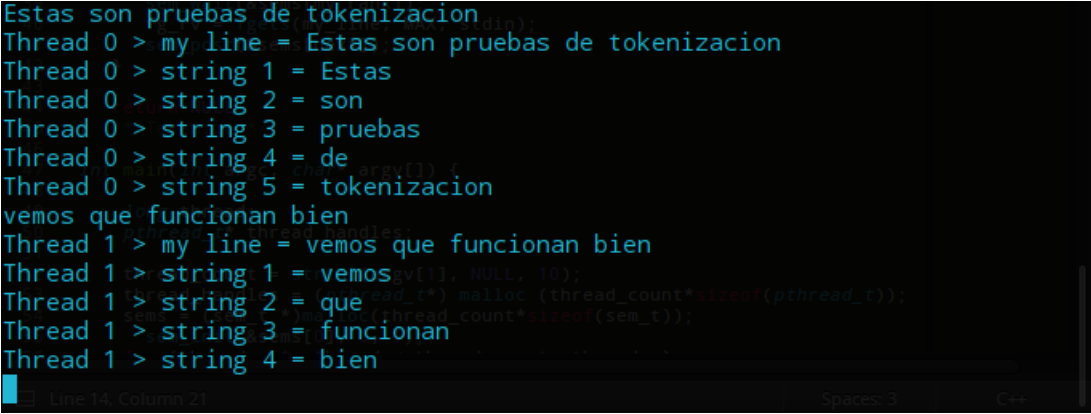
```

char my_line[MAX+1];
char *my_string, *saveptr;

/* Force sequential reading of the input */
sem_wait(&sems[my_rank]);
fg_rv = fgets(my_line, MAX, stdin);
sem_post(&sems[next]);
while (fg_rv != NULL) {
    printf("Thread %ld > my line = %s", my_rank, my_line);

    count = 0;
    my_string = strtok_r(my_line, " \t\n", &saveptr);
    while ( my_string != NULL ) {
        count++;
        printf("Thread %ld > string %d = %s\n", my_rank, count, my_string);
        my_string = strtok_r(NULL, " \t\n", &saveptr);
    }
    sem_wait(&sems[my_rank]);
    fg_rv = fgets(my_line, MAX, stdin);
    sem_post(&sems[next]);
}

```



```

Estas son pruebas de tokenizacion
Thread 0 > my line = Estas son pruebas de tokenizacion
Thread 0 > string 1 = Estas
Thread 0 > string 2 = son
Thread 0 > string 3 = pruebas
Thread 0 > string 4 = de
Thread 0 > string 5 = tokenizacion
vemos que funcionan bien
Thread 1 > my line = vemos que funcionan bien
Thread 1 > string 1 = vemos
Thread 1 > string 2 = que
Thread 1 > string 3 = funcionan
Thread 1 > string 4 = bien

```

Figura 1.10: Ejecución del programa tokenizar