Universidad Nacional de San Agustín

Facultad de Producción y Servicios

Escuela Profesional de Ciencia de la Computación

Curso :

Algoritmos Paralelos

Tema :

Programación de Memoria Compartida con Pthreads - Informe de Laboratorio

Docente :

Mgter. Alvaro Henry Mamani Aliaga

Alumno :

Vidal Antonio Soncco Merma

AREQUIPA - PERÚ

2017

1 Multiplicación Matríz - Vector usando Pthreads

Si $A = (a_{ij})$ es una matriz de $m \times n$ y x es un vector con una columna $n-$dimensional, entonces el producto de matriz-vector $Ax = y$ es vector de columna $m-$dimensional donde el componente ith y_i es obtenido encontrando el producto punto de la $i-$th fila de A con x.

Código :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable:  accesible to all threads */
int thread_count;
long long n;
double sum;

void* Thread_sum(void* rank);   /* Thread function */

int main(int argc, char* argv[]) {
        long             thread;        /* Use long in case of a 64-bits system */
        pthread_t*       thread_handles;

        /* Gets number of threads from command line */
        thread_count = strtol(argv[1], NULL, 10);

        thread_handles = malloc (thread_count*sizeof(pthread_t));

        printf("Ingrese el valor de n\n");
        scanf("%lld", &n);

        for (thread = 0; thread < thread_count; thread++)
                pthread_create(&thread_handles[thread], NULL, Thread_sum, (void*)
                    thread);

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

        sum = 4.0*sum;
        printf("Con n = %lld terminos,\n", n);
        printf("Nuestro estimado de pi = %f\n", sum);

        free(thread_handles);
        return 0;
}  /* main */

void* Thread_sum(void* rank) {
        long my_rank = (long) rank;      /* Use long in case of a 64-bits system */
        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;

        if (my_first_i % 2 == 0)         /* my_first_i is even */
                factor = 1.0;
        else                                             /* my_first_i is odd */
                factor = -1.0;

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

        return NULL;
}  /* Thread_sum */
```

2 Función para el Cálculo de Pi π

Podemos intentar paralelizar esto de la misma manera que paralelizamos el programa de multiplicación vector-matriz: dividir las iteraciones en el bucle *for* entre los hilos y hacer *sum* una variable compartida.

Código :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable:  accesible to all threads */
int thread_count;
long long n;
double sum;

void* Thread_sum(void* rank);   /* Thread function */

int main(int argc, char* argv[]) {
        long            thread;         /* Use long in case of a 64-bits system */
        pthread_t*      thread_handles;

        /* Gets number of threads from command line */
        thread_count = strtol(argv[1], NULL, 10);

        thread_handles = malloc (thread_count*sizeof(pthread_t));

        printf("Ingrese el valor de n\n");
        scanf("%lld", &n);

        for (thread = 0; thread < thread_count; thread++)
                pthread_create(&thread_handles[thread], NULL, Thread_sum, (void*)
                    thread);

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

        sum = 4.0*sum;
        printf("Con n = %lld terminos,\n", n);
        printf("Nuestro estimado de pi = %f\n", sum);

        free(thread_handles);
        return 0;
}  /* main */

void* Thread_sum(void* rank) {
        long my_rank = (long) rank;     /* Use long in case of a 64-bits system */
        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;

        if (my_first_i % 2 == 0)        /* my_first_i is even */
                factor = 1.0;
        else                                            /* my_first_i is odd */
                factor = -1.0;

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

        return NULL;
```

```
}  /* Thread_sum */
```

3 Busy-Waiting y Mutexes

3.1 Busy-Waiting

Es una técnica donde un proceso repetidamente verifica una condición, tal como esperar una entrada de teclado o si el ingreso a una sección crítica está habilitado. Un enfoque simple que no implica ningún nuevo concepto es el uso de una variable *flag*.

Suma Global Pthreads con Busy-Waiting

```
void* Thread_sum(void* rank) {
        long my_rank = (long) rank;      /* Use long in case of a 64-bits system */
        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;

        if (my_first_i % 2 == 0)         /* my_first_i is even */
                factor = 1.0;
        else                                              /* my_first_i is odd */
                factor = -1.0;

        for (i = my_first_i; i < my_last_i; i++, factor = -factor){
                while (flag != my_rank);
                sum += factor/(2*i + 1);
                flag = (flag + 1) % thread_count;
        }

        return NULL;
}
```

Suma Global Pthreads con Secciones Críticas después del bucle

```
void* Thread_sum2(void* rank) {
        long my_rank = (long) rank;      /* Use long in case of a 64-bits system */
        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)         /* my_first_i is even */
                factor = 1.0;
        else                                              /* my_first_i is odd */
                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;
        flag = (flag + 1) % thread_count;

        return NULL;
}  /* Thread_sum */
```

3.2 Mutexes

Un mutex es un tipo especial de variable que, junto con un par de funciones especiales, se puede utilizar para restringir el acceso a una sección crítica a un solo hilo a la vez. Por lo tanto, un mutex se puede utilizar para garantizar que un hilo "excluye" todos los demás hilos mientras se ejecuta la

sección crítica. Por lo tanto, el mutex garantiza el acceso mutuamente exclusivo a la sección crítica.

Función de suma global que utiliza un mutex

```
void* Thread_sum(void* rank) {
        long my_rank = (long) rank;      /* Use long in case of a 64-bits system */
        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)         /* my_first_i is even */
                factor = 1.0;
        else                                          /* my_first_i is odd */
                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;
}  /* Thread_sum */
```

3.3 Tiempos de ejecución

Tiempos de ejecución (en segundos) del Programas Usando $n = 10^8$ Términos en un sistema con dos procesadores de cuatro núcleos

Hebras	Busy-Wait	Mutex
1	1.548856	1.054495
2	6.984196	5.285079
3	1.015402	5.018480

Table 1: Programa del Calculo de Pi

References

[1] http://rubmarin.galeon.com/sisd.htm

[2] http://www.academia.edu/18404632/Taxonomías_de_Flynn

[3] https://es.slideshare.net/cramex/programacin-paralela-conceptos-y-diseo-de-sistemas-distribuido