Algoritmos Paralelos

Grimaldo José Dávila Guillén

April 17, 2017

Contents

1 Matriz-Vector usando PThreads

En esta prueba se realizó una multiplicación tanto en serie como paralela para obtener los resultados de la tabla 1.

Primero empezaremos describiendo el código de la figura 1 en la cual podemos ver el código paralelo de la función multiplicación matriz-vector. De la línea 76 a la 78 podemos ver como es que segmentamos el área de la matriz para poder multiplicar solamente esa área con la hebra correspondiente. De la línea 80 a 87 podemos ver el bucle en el cual se realiza la multiplicación solamente del segmento que le corresponde a dicha hebra con el ID rank que es pasado como parámetro.

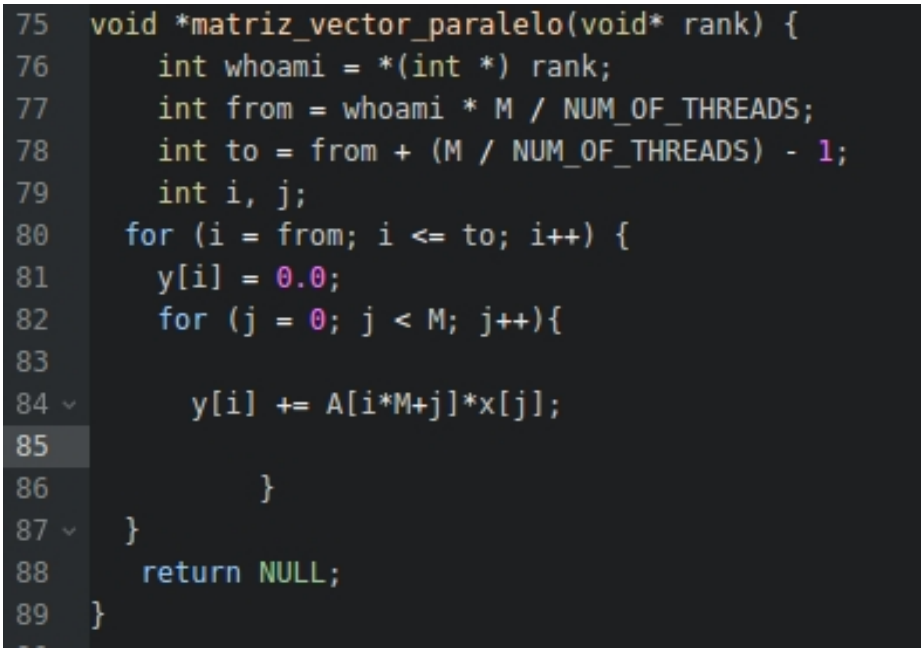

Figure 1: Multiplicación con pthreads.

En la figura 2 vemos el main del programa en el cual podemos ver la línea 102 es donde se crea el espacio de memoria para las hebras correspondiendo para este programa y en las líneas 113 y 114 se crean las hebras juntos con las funciones que realizaran y su respectivo ID.

```
94  int main(int argc, char* argv[]){
95  |
96      M = atoi(argv[1]);
97      N = atoi(argv[2]);
98      NUM_OF_THREADS = atoi(argv[3]);
99
100   int i;
101
102   pthread_t* threads = (pthread_t *)malloc(NUM_OF_THREADS * sizeof(pthread_t));
103
104   A = (float *) malloc(M * N * sizeof(float));
105   x = (float *) malloc(N * sizeof(float));
106   y = (float *) malloc(M * sizeof(float));
107
108   LLenar_matriz(A, M, N);
109   LLenar_vector(x, N);
110
111   // Creamos los threads
112
113   for(i = 0; i < NUM_OF_THREADS; i++)
114     pthread_create(&threads[i], NULL,matriz_vector_paralelo, &i);
115
116
117   for (i = 0; i < NUM_OF_THREADS; i++)
118     pthread_join(threads[i], NULL);
119
120   free(A);
121   free(x);
122   free(y);
123   free(threads);
124 }
```

Figure 2: Creación de threads y utilización.

En la figura 3 podemos ver los resultados de una prueba donde obtenemos el resultados de los tiempo tanto en serie com oen paralelos de las funciones multiplicar_matriz_vector.

```
grimaldo@grimaldo-Lenovo-Z50-70:~/Documentos/computer_science/AlgoritmosParalelo
s$ ./ejemplo 8000 8000 1
TEST OK
SIZE: 8000 x 8000 AND 1 THREADS
TIEMPO CONC.:238 mseg
TIEMPO SERIAL:238 mseg

-------------------------------------------------------
```

Figure 3: Resultados en consola.

En la tabla 1 podemos ver que mientras mas hebras hay trabajando sincronizadamente se obtiene una mejora tanto en tiempo como en eficiencia esto es debido a que las tareas son repartidas en varias hebras estas se pueden realizan de una manera mas rápida.

Tamaño de Matriz						
	8 000 000 x 8		8000 x 8000		8 x 8 000 000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.312	1.000	0.238	1.000	0.386	1.000
2	0.235	0.843	0.157	0.92	0.284	0.652
4	0.127	0.689	0.096	0.79	0.317	0.261

Table 1: Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

2 Cálculo de PI

Para el cálculo de PI utilizaremos dos métodos vistos en clase. El primero es usando mutex y el siguiente con busy waiting.

2.1 Usando Mutex

Una forma de evitar que varias threads modifiquen un mismo dato a las vez es usar mutex. Esto bloqueara dicha variable todo el tiempo que una thread la este utilizando así ya no correremos el riesgo de tener valores erróneos en nuestros cálculos.
En las líneas 72 y 74 de la figura 4 podemos ver como bloqueamos con mutex la variable global SUM para que así solo pueda ser utilizada en ese momento por la thread actual.

```
53  void*  Thread_sum_mutex(void * rank)
54  {
55    int my_rank = *((int*)rank);
56    double factor;
57    long long i;
58    long long my_n=N/NUM_OF_THREADS;
59    long long my_first_i=my_n*my_rank;
60    long long my_last_i=my_first_i+my_n;
61    double my_sum=0.0;
62
63    if((my_first_i%2) ==0)
64       factor=1.0;
65    else
66       factor=-1.0;
67
68    for(i=my_first_i;i<my_last_i;i++,factor=-factor){
69      my_sum+=factor/(2*i+1);
70    }
71
72    pthread_mutex_lock(&mutex);
73    sum+=my_sum;
74    pthread_mutex_unlock(&mutex);
75
76  }
```

Figure 4: Global sum function that uses a mutex

2.2 Usando Busy Waiting

Otra forma de evitar que varias threads modifiquen un mismo dato a las veces usar el método de busy waiting pero no es una manera muy eficiente debido a que el hecho de utilizar un bucle dando indefinidas iteraciones causa un desperdicio de procesamiento y recursos del computador a pesar de que nos daría el mismo resultado que usar mutex.

En las líneas 44 46 de la figura 5 podemos ver como simulamos un mutex utilizando un bucle while indefinido mientras espera que flag sea igual al thread correspondiente.

```
26  void*  Thread_sum_busy_W(void * rank)
27  {
28    int my_rank = *((int*)rank);
29    double factor;
30    long long i;
31    long long my_n=N/NUM_OF_THREADS;
32    long long my_first_i=my_n*my_rank;
33    long long my_last_i=my_first_i+my_n;
34    double my_sum=0.0;
35
36    if((my_first_i%2) ==0)
37       factor=1.0;
38    else
39       factor=-1.0;
40
41    for(i=my_first_i;i<my_last_i;i++,factor=-factor){
42      my_sum+=factor/(2*i+1);
43    }
44    while(flag!=my_rank);
45    sum+=my_sum;
46    flag=(flag+1)%NUM_OF_THREADS;
47
48  }
```

Figure 5: Global sum function with critical section after loop

En la figura 6 podemos ver los resultados en consola del calculo de PI usando mutex utilziando 64 threads y un N= $10\hat{8}$.

```
grimaldo@grimaldo-Lenovo-Z50-70:~/Documentos/computer_science/AlgoritmosParalelo
s$ ./pi 100000000 64
TIEMPO CONC.:547 mseg
```

Figure 6: cálculo de PI usando mutex

Threads	Busy-Wait	Mutex
1	0.533	0.563
2	0.538	0.550
4	0.574	0.520
8	0.575	0.571
16	0.529	0.579
32	0.569	0.535
64	0.548	0.537

Table 2: Table 4.1 Run times of n programs using n equal to 100000000 terms