Informe 04 - Algoritmos Paralelos

Lorena Xiomara Castillo Galdos

Abril 2017

Contents

1 Multiplicación Matriz-Vector

En la multiplicación de una matriz por un vector si $A = (a_{ij})$ es una matriz de mxn y $x = (x_0, x_1, ..., x_{n-1})^T$ es un vector columna n-dimensional, entonces el producto matriz-vector $Ax = y$ es un vector columna m-dimensional, $y = (y_0, y_1, y_{m-1})^T$ en el cual el i-ésimo componente y_i es obtenido al encontrar el producto punto de la i-ésima fila de A con x. Figura 1

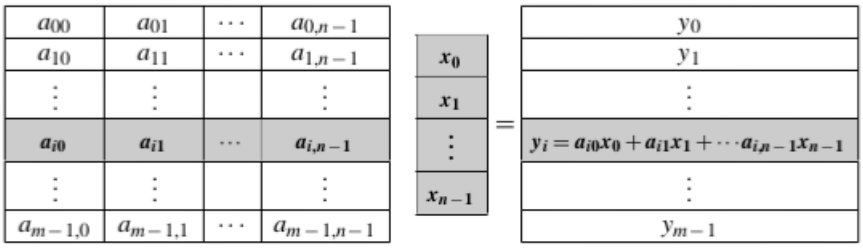

Figure 1: Multiplicación matriz-vector.

El código del programa se muestra a continuación

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <pthread.h>
#include <sys/time.h>

int NUM_OF_THREADS, M, N;
float* A;
float* x;
float* y;
float* y_s;
pthread_mutex_t mutex;


long Tiempo_segundos(timeval inicio, timeval fin)
{
   long seg, useg;
   seg  = fin.tv_sec  - inicio.tv_sec;
   useg = fin.tv_usec - inicio.tv_usec;
   return ((seg) * 1000 + useg/1000.0) + 0.5;
}

void Llenar_matriz(float A[], int f, int c)
{
   int i, j;
   for (i = 0; i < f; i++)
   {
      for (j = 0; j < c; j++)
      {
         A[i * c + j] = (rand() % 100); //
```

```
      }
   }
}

void Llenar_vector(float x[], int f)
{
   int i;
   for (i = 0; i < f; i++)
   {
      x[i] = (rand() % 100); //
   }
}

void Print_matriz(float A[], int f, int c)
{
   int i, j;
   for (i = 0; i < f; i++)
   {
      for (j = 0; j < c ; j++)
      {
           printf("%f  ", A[i*c + j]);
      }
      printf("\n");
   }
}

void Print_vector(float x[], int f)
{
   int i;
   for (i = 0; i < f; i++)
   {
      printf("%f  ", x[i]);
   }
   printf("\n");
}

//P
void *Mat_vect_mult(void* rank)
{
   int my_rank = *(int *) rank;
   int de_t = my_rank * M / NUM_OF_THREADS;
   int a_t = de_t + (M / NUM_OF_THREADS) - 1;
   int i, j;
   for (i = de_t; i <= a_t; i++)
   {
      y[i] = 0.0;
      for (j = 0; j < M; j++)
      {
         y[i] += A[i*M+j]*x[j];
      }
   }
   return NULL;
}

//S
void _Mat_vect_mult(void)
{
```

```
   int i,j;
   for(i = 0; i < M; i++)
   {
      y_s[i] = 0.0;
      for(j = 0; j < N; j++)
      {
         y_s[i] += A[i * N + j] * x[j];
      }
   }
}


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

    timeval inicio;
    timeval fin;

    int fin_p, fin_s ; // long
    if(argc != 4)
    {
        fputs("Parametros son M N y NUM_OF_THREADS\n", stdout);
        return EXIT_FAILURE;
    }
    else
    {
       M = atoi(argv[1]);
       N = atoi(argv[2]);
       NUM_OF_THREADS = atoi(argv[3]);
    }

    //srand(time(NULL));

    int i, _error;

    pthread_t* threads = (pthread_t *)malloc(NUM_OF_THREADS * sizeof(pthread_t));

    A   = (float *) malloc(M * N * sizeof(float));
    x   = (float *) malloc(N * sizeof(float));
    y   = (float *) malloc(M * sizeof(float));
    y_s = (float *) malloc(M * sizeof(float));

    Llenar_matriz(A, M, N);

    Llenar_vector(x, N);

    gettimeofday(&inicio, 0);

    for(i = 0; i < NUM_OF_THREADS; i++)
    {
      _error = pthread_create(&threads[i], NULL, Mat_vect_mult, &i);
      if (_error)
      {
        fprintf(stderr, "error: pthread_create, rc: %d\n", _error);
        return EXIT_FAILURE;
      }
    }
    for (i = 0; i < NUM_OF_THREADS; i++)
```

```
    {
        pthread_join(threads[i], NULL);
    }
//P
    gettimeofday(&fin, 0);
    fin_p = Tiempo_segundos(inicio, fin);
//S
    gettimeofday(&inicio, 0);
    _Mat_vect_mult();

    gettimeofday(&fin, 0);
    fin_s = Tiempo_segundos(inicio, fin);


    fprintf(stdout, "SIZE: %d x %d AND %d THREADS\n", M, N, NUM_OF_THREADS);
    fprintf(stdout, "TIEMPO_P_:%d_mseg\n", fin_p);
    fprintf(stdout, "TIEMPO_S_:%d_mseg\n", fin_s);

    free(A);
    free(x);
    free(y);
    free(y_s);
    free(threads);
    pthread_mutex_destroy(&mutex);

}
```

El resultado se muestra en la Figura 2.

```
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./matriz_vector 8000 8000 1
SIZE: 8000 x 8000 AND 1 THREADS
TIEMPO P :241 mseg
TIEMPO S :236 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./matriz_vector 8000 8000 2
SIZE: 8000 x 8000 AND 2 THREADS
TIEMPO P :135 mseg
TIEMPO S :234 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./matriz_vector 8000 8000 4
SIZE: 8000 x 8000 AND 4 THREADS
TIEMPO P :147 mseg
TIEMPO S :234 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$
```

Figure 2: Resultados multiplicación matriz-vector.

Los tiempos y eficiencia de la multiplicación matriz-vector se muestran en la tabla 1.

Table 1: Tabla de tiempos y eficiencia de multiplicación matriz-vector.

	Dimensión de la Matriz					
	8000000 x 8		8000 x 8000		8 x 8000000	
Threads	Tiempo	Eff	Tiempo	Eff	Tiempo	Eff
1	0.323	1.000	0.241	1.000	0.342	1.000
2	0.212	0.836	0.135	0.901	0.360	0.712
4	0.119	0.709	0.147	0.771	0.278	0.265

2 Cálculo de PI

2.1 Busy Waiting y Mutex

Al implementar una barrera usando $busy-waiting$ y $mutex$ usamos un contador compartido protegido por el $mutex$. Cuando el contador indica que cada hebra ha entrado a la sección crítica, las hebras pueden dejar el bucle $busy - wait$. Necesitamos usar una variable contador por cada instancia de una barrera.[1]

Dado que una hebra que está en $busy - waiting$ puede continuamente usar el CPU, $busy - waiting$ no es generalmente una solución ideal al problema de limitar el acceso a una sección crítica, $Mutex$ es una abreviación de exclusión mutua, y $mutex$ es un tipo especial de variable que, junto con un par de funciones especiales, puede ser usado para garantizar que una hebra "excluya" a las demás cuando ejecuta una sección crítica. Por lo tanto $mutex$ garantiza acceso exclusivo mutuo a la sección crítica.

A continuación se muestra el código del cálculo de pi utilizando $busy - waiting$

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <pthread.h>
#include <sys/time.h>

int NUM_OF_THREADS, N;
pthread_mutex_t mutex;
double sum=0.0;
int flag=0;

long Tiempo_total(timeval inicio, timeval fin)
{
  long seg, useg;
  seg  = fin.tv_sec  - inicio.tv_sec;
  useg = fin.tv_usec - inicio.tv_usec;
  return ((seg) * 1000 + useg/1000.0) + 0.5;
}

//BW
void* Pi_BW(void * rank)
{
```

```
  int my_rank = *((int*)rank);
  double factor;
  long long i;
  long long my_n = N/NUM_OF_THREADS;
  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);
  }
  while(flag!=my_rank);
  sum += my_sum;
  flag = (flag+1) \% NUM_OF_THREADS;
}

int main(int argc, char* argv[])
{
   N = atoi(argv[1]);
   NUM_OF_THREADS = atoi(argv[2]);
   timeval inicio;
   timeval fin;
   long fin_p;
   srand(time(NULL));
   int i;
   pthread_t* threads = (pthread_t *)malloc(NUM_OF_THREADS * sizeof(pthread_t));
   gettimeofday(&inicio, 0);
   for(i = 0; i < NUM_OF_THREADS; i++)
   {
      pthread_create(&threads[i], NULL,Pi_BW,&i);
      pthread_join(threads[i], NULL);
   }
   gettimeofday(&fin, 0);
   fin_p = Tiempo_total(inicio, fin);
   fprintf(stdout, "TIEMPO_P_:_\%d_mseg\n",(int) fin_p);
   free(threads);
}
```

El resultado se muestra en la Figura 3.

```
lorena@Lorena-pc:~/Documentos/Paralelos/t$ g++ pi.c -o pi -lpthread
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 1
TIEMPO P : 531 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 2
TIEMPO P : 532 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 4
TIEMPO P : 527 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 8
TIEMPO P : 546 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 16
TIEMPO P : 542 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 32
TIEMPO P : 530 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 64
TIEMPO P : 539 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$
```

Figure 3: Resultados de cálculo de pi con busy-waiting.

A continuación se muestra el código del cálculo de pi utilizando *mutex*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <pthread.h>
#include <sys/time.h>

int NUM_OF_THREADS, N;
pthread_mutex_t mutex;
double sum=0.0;
int flag=0;

long Tiempo_total(timeval inicio, timeval fin)
{
  long seg, useg;
  seg  = fin.tv_sec  - inicio.tv_sec;
  useg = fin.tv_usec - inicio.tv_usec;
  return ((seg) * 1000 + useg/1000.0) + 0.5;
}

//Mtx
void* Pi_Mtx(void * rank)
{
  int my_rank = *((int*)rank);
  double factor;
  long long i;
  long long my_n = N/NUM_OF_THREADS;
  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);
}

int main(int argc, char* argv[])
{
   N = atoi(argv[1]);
   NUM_OF_THREADS = atoi(argv[2]);
   timeval inicio;
   timeval fin;
   long fin_p;
   srand(time(NULL));
   int i;
   pthread_t* threads = (pthread_t *)malloc(NUM_OF_THREADS * sizeof(pthread_t));
   gettimeofday(&inicio, 0);
   for(i = 0; i < NUM_OF_THREADS; i++)
   {
      pthread_create(&threads[i], NULL, Pi_Mtx,&i);
      pthread_join(threads[i], NULL);
   }
   gettimeofday(&fin, 0);
   fin_p = Tiempo_total(inicio, fin);
   fprintf(stdout, "TIEMPO_P_:_%d_mseg\n",(int) fin_p);
   free(threads);
}
```

El resultado se muestra en la Figura 4.

```
lorena@Lorena-pc:~/Documentos/Paralelos/t$ g++ pi.c -o pi -lpthread
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 1
TIEMPO P : 541 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 2
TIEMPO P : 564 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 4
TIEMPO P : 525 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 8
TIEMPO P : 546 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 16
TIEMPO P : 531 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 32
TIEMPO P : 529 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$ ./pi 100000000 64
TIEMPO P : 552 mseg
lorena@Lorena-pc:~/Documentos/Paralelos/t$
```

Figure 4: Resultados de cálculo de pi con mutex.

En la tabla 2 se muestran los resultados del cálculo de pi con $busy-waiting$ y $mutex$.

Table 2: Tiempos de ejecución en segundos con $n = 10^8$.

Threads	Busy-Wait	Mutex
1	0.531	0.541
2	0.532	0.564
4	0.527	0.525
8	0.546	0.546
16	0.542	0.531
32	0.530	0.529
64	0.539	0.552

References

[1] Peter Pacheco. *An introduction to parallel programming.* Elsevier, 2011.