Laboratorio Algoritmos Paralelos

André Mogrovejo Martínez

Mayo 2019

1. Introducción

En este trabajo se busca replicar las tablas del capítulo 5 de [1]. Con el fín de sacar conclusiones propias respecto a los métodos de manejo en las secciones críticas de cada algoritmo y entender la diferencia entre las distintas formas de implementación de estos métodos.

2. Multiplicación Matrix-Vector

Código:

```
1 void Omp_matriz_vector(double *A,double *x,double  *y,LLi m,LLi n,int thread_count)
2 {
3   LLi i,j;
4   double start, finish, tiempoF;
5   start = omp_get_wtime();
6   #pragma omp parallel for num_threads(thread_count) default(none) private(i,j) shared(A,x,y,m,n)
7   for(i=0;i<m;i++)
8   {
9     y[i] = 0.0;
10    for(j=0; j < n; j++)
11    {
12      y[i] += A[i*n+j]*x[j];
13    }
14  }
15  finish = omp_get_wtime();
16  tiempoF = finish - start;
17  printf("Tiempo de ejecucion es: %e\n",tiempoF);
18 }
```

2.1. Resultados

Matrix			
thread	8000000x8	8000x8000	8x8000000
1.000000	0.274518	0.264664	0.248658
2.00	0.274061	0.258864	0.269361
4.00	0.275367	0.248516	0.267842
8.00	0.274271	0.248432	0.260086
16.00	0.275379	0.248920	0.248826
32.00	0.275294	0.248977	0.248826
64.00	0.274317	0.248697	0.247861

Figura 1: Tabla de tiempos de Matrix-Vector Multiplication

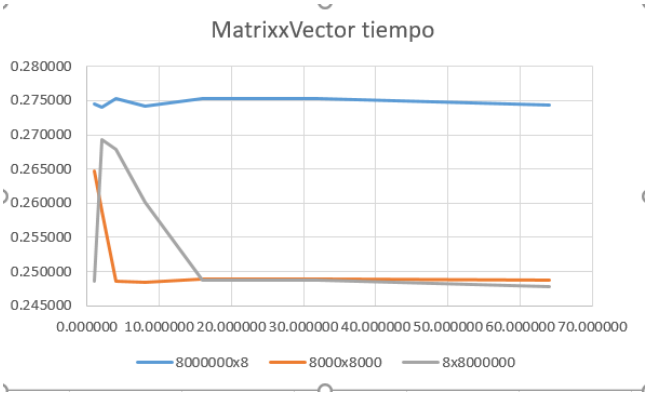

Figura 2: Grafica de tiempos de Matrix-Vector Multiplication

thread	8000000x8	8000x8000	8x8000000
1.000000	1.000000	1.000000	1.000000
2.00	0.998335	0.978084	1.083260
4.00	1.003093	0.938987	1.077148
8.00	0.999100	0.938671	1.045960
16.00	1.003137	0.940514	1.000674
32.00	1.002828	0.940730	1.000676
64.00	0.999267	0.939670	0.996796

Figura 3: Tabla de eficiencias de Matrix-Vector Multiplication

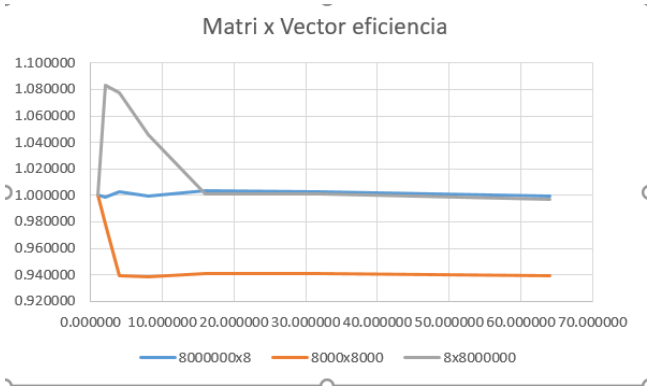

Figura 4: Grafica de eficiencias de Matrix-Vector Multiplication

2.2. Explicación de los resultados

En nuestro caso no existe dependencias en bucle en el bucle externo, ya que la Matriz y el vector no se actualizan y la iteración i solo actualiza el resultado. Por lo tanto, podemos paralelizar esto dividiendo las iteraciones en el bucle externo entre los hilos.

- Para las entradas de 8 000 000 x 8 se requiere un número un tanto mas elevado de tiempo respecto al caso de 8 000 x 8000.
- Para las entradas de 8 000 000 x 8 se requiere un número un tanto mas elevado de tiempo respecto al caso de 8 000 x 8000.

Esto se dá debido al diseño propio de la memoria cache, que por principios de localidad tanto espacial como temporal, hay una diferencia en la velocidad de acceso a la caché.

Podemos inferir que cual se toma las entradas de 8 000 000 x 8, existen un gran número de *fallas de escritura*.

Por otro lado en el caso de 8 x 8 000 000 hay demasiadas *fallas de lectura*, que ocurre cuando una variable pensada, resulta no estar en memoria caché y es necesario buscarla en memoria principal.

Se puede dar un aproximado en los valores de eficiencia, donde el caso 8 x 8 000 000 aproximadamente un 20 % menos eficiente respecto al caso de una matriz cuadrada. Por otro lado el caso de 8 000 000 x 8 tiene una eficiencia aun peor dandonos un valor aproximado del 50 %.

3. Odd-Even sort

3.1. OE 1

Código:

```
void odd_even_sort_OpenMp1(double *a,int n,int thread_count)
{
  int phase,i,temp;
  double start, finish, tiempoF;
  start = omp_get_wtime();
  #ifdef DEBUG
  char title[100];
  #endif

  for(phase = 0; phase < n; phase++)
  {
    if((phase \% 2)==0)
    {
      #pragma omp parallel for num_threads(thread_count) default(none) shared(a,n) private(i,temp)
      for(i = 1; i < n; i+=2)
      {
        if(a[i-1]>a[i])
        {
          temp= a[i];
          a[i] = a[i-1];
          a[i-1] = temp;
        }
      }
    }
    else {
      #pragma omp parallel for num_threads(thread_count) default(none) shared(a,n) private(i,temp)
      for(i = 1; i < n-1; i+=2)
      {
        if(a[i] > a[i+1])
        {
          temp= a[i];
          a[i] = a[i+1];
          a[i+1] = temp;
        }
      }
    }
    #ifdef DEBUG
    sprintf(title,"After phase \%d",phase);
    Print_list(a,n,title);
    #endif

  }
  finish = omp_get_wtime();
  tiempoF = finish - start;
  printf("Tiempo de ejecucion es: \%e\n",tiempoF);

}
```

3.2. OE 1

Código:

```
void odd_even_sort_OpenMp2(double *a,int n,int thread_count)
{
  int phase,i,temp;
  double start, finish, tiempoF;
  start = omp_get_wtime();
  #pragma omp parallel for num_threads(thread_count) default(none) shared(a,n) private(i,temp,phase)
  for(phase = 0; phase < n; phase++)
  {
    if((phase \% 2)==0)
    {
      #pragma omp parallel for

      for(i = 1; i < n; i+=2)
      {

        if(a[i-1]>a[i])
        {
          temp= a[i];
          a[i] = a[i-1];
          a[i-1] = temp;
        }
      }
    }
    else {
      #pragma omp parallel for
      for(i = 1; i < n-1; i+=2)
      {
        if(a[i] > a[i+1])
        {
          temp= a[i];
          a[i] = a[i+1];
          a[i+1] = temp;
        }
      }
    }

  }
  finish = omp_get_wtime();
  tiempoF = finish - start;
  printf("Tiempo de ejecucion es: \%e\n",tiempoF);

}
```

3.3. Resultados

thread	Odd-even1	Odd-even2
2.00	1.122019	2.069337
4.00	1.137670	0.851341
8.00	1.122405	0.412278
16.00	1.138762	0.317000
32.00	1.108479	0.222885
64.00	1.126100	0.173254

Figura 5: Tabla de tiempos de Odd-Even comparativa

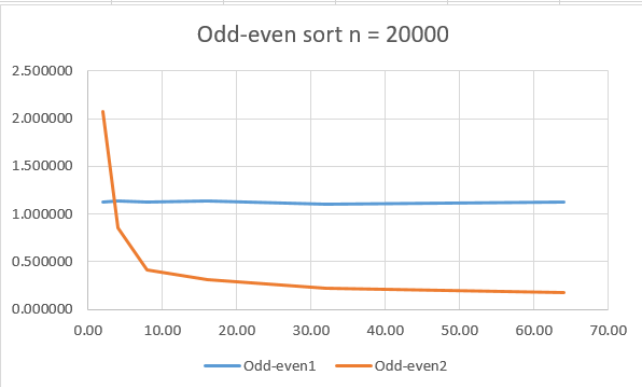

Figura 6: Grafica de tiempos de Odd-Even comparativa

3.4. Explicación de los resultados

Cuando utilizamos dos o mas subprocesos, la versión que usa dos para las directivas es al menos un 17 % mas rapida que la version que usa dos for paralelos para las directivas, por lo que para este sistema vale la pena el leve esfuerzo que implica realizar el cambio.

Como los resultados nos indican, la segunda implementación del Odd-Even tiene una notable diferencia respecto a la primera, donde a medida que aumentamos el número de threads el tiempo de ejecución en este código disminuye.

Este comportamiento es debido a que podemos unir nuestro equipo de hilos de conteo de hilos antes del bucle externo con una directiva paralela. Luego, en lugar de forjar un nuevo equipo de subprocesos con cada ejecución de uno de los bucles internos, usamos una directiva for, que le dice a OpenMP que paralice el bucle for con el equipo de subprocesos existente.

La directiva for a diferencia del directivo paralelo, no se divide en ningún hilo. Utiliza cualquier hilo que ya haya sido bifurcado en el bloque paralelo que lo contiene. Hay una barrera implıcita al final del bucle. Los resultados del código, la lista final, serán los mismos que los resultados obtenidos del código original paralelizado.

Referencias

[1] P. Pacheco, An Introduction to Parallel Programming, 2011