



Trabajo Encargado

Computación paralela

Métodos de Ordenamiento Usando OpenMP



Encargado: RENZO APAZA CUTIPA

Autores:

Valeria Jara Sanca

Milagros Arocutipa Cusacani

Franco Champio Mamani

mcs

**INTRODUCCION**

El siguiente trabajo se propone implementa la computación paralela a los algoritmos de ordenamiento, para esto se hicieron tanto para el ámbito lineal o también llamado tradicional como así también la parte paralela, mas se caracteriza el trabajo de una implementación de algoritmos de ordenamiento en paralelo, pero a su vez también se incluyó la comparación para poder observar la eficiencia de la implementación en paralelo, para su ves observar de la veracidad de este y hacer comparaciones usando la librería time.h en nuestro algoritmo, se implementó un menú para poder controlar y manipular cada ordenamiento que en total son 4 es decir que tenemos los siguientes algoritmos de ordenamiento la primera que sería burbujeo la segunda Cocktail Sort, el tercero que se implementó Shell Sort y por ultimo tenemos el algoritmo de ordenamiento Insertion Sort,

**IMPLEMENTACION DE LOS ALGORTIMOS DE ORDENAMIENTO**

**Plataforma y Librería**

Entorno y IDE

Los siguientes algoritmos ya mencionados se implementaron en el lenguaje C/C++ y en IDE de Codeblok ya que se ha estado trabajando las prácticas en dicho IDE y lenguaje también puedo haber implementado en diversos IDEs preferencial ya sea Visual studio, entre otros, se trabaja usando el procesamiento en paralelo de OpenMP, como mejor alternativa también se puedo trabajar como (CUDA , MPI) como se observó en las últimas sesiones, pero para fines prácticos y un amplio compatibilidad y el cubrimiento de soporte de trabajo con el procesamiento de OpenMP. También se incluyo la insltacion previa de la librería nativa OpenMP (#include <omp.h>.

Desarrollo

El programa comienza cargando un vector desordenado, utilizando las funciones para la generación de números pseudo-aleatorios de la librería stdlib, inicializando siempre la generación de la semilla con la función:

srand(2);

para que todas las ejecuciones posean el mismo vector, para lograr un informe lo más objetivo posible. Para setear la cantidad de threads deseados, se utilizó la función provista por OpenMP, que recibe por parámetro la cantidad a crear:

omp\_set\_num\_threads(CANTIDAD\_THREADS);

Luego el programa solicitará por pantalla la selección de una de las 4 opciones, que representan a los distintos algoritmos de ordenamiento; una vez seleccionada; el programa ejecutará primero un ordenamiento lineal, y luego en paralelo; indicando los tiempos de cada uno. Para llevar a cabo dicho objetivo, se utilizó la librería time, donde al comienzo de cada ordenamiento se declaró una variable de tipo clock\_t: clock\_t t; t = clock();

y al finalizar dicho ordenamiento, se obtuvo la diferencia, para luego mostrarla por pantalla tanto en ticks del reloj como en segundos, usando la constante CLOCKS\_PER\_SEC:

t = clock() - t;

cout<<"Ordenamiento realizado.\n";

cout<<"Tardo "<<t<<" ticks del reloj. ( "<< ((float)t)/CLOCKS\_PER\_SEC<<" segundos)";

Luego de cada set de ordenamientos, es posible ver determinada posición del vector, para corroborar que el mismo fue correctamente realizado.

**Algoritmos de ordenamiento implementadas**

Para el trabajo realizado teniendo como 4 métodos de ordenamientos las cuales son:

1. Ordenamiento por burbujeo
2. Cocktail Sort
3. Shell Sort
4. Insertion Sort

Código de implementación

1. burbujeo

Performance para el peor caso: О(*n*2)

Performance para el mejor caso: O(*n*)

Complejidad: O(n²)

* 1. Algoritmo de ordenamiento lineal o tradicional

#include <iostream>  
//Ordenamiento burbujeo lineal  
 using namespace std;  
 int main()  
 {  
  
 int i,j,iAux;  
 int v2[] = {3,2,1,5,4}; //vector en desordenado  
 int iPaso=1;  
  
 bool bandera;  
 do  
 {  
 bandera=true;  
 for(i=0;i<5-iPaso;i++)  
 {  
 if(v2[i]>v2[i+1])  
 {  
 iAux = v2[i];  
 v2[i] = v2[i+1];  
 v2[i+1] = iAux;  
 bandera=false;  
 }  
 }  
 iPaso++;  
 }while(!bandera);  
  
 cout<<"ordenamiento de burbujeo ";  
  
//imprimiendo datos  
 for(i=0;i<5;i++){  
 cout<<v2[i];  
 }  
 }

* 1. Algoritmo de ordenamiento paralelo

#include <iostream>

#include <omp.h>  
using namespace std;  
int main()  
 {  
int fase;  
int tmp;  
int v2[] = {3,2,1,5,4};  
int i,j;  
  
#pragma omp parallel default(none) shared(v2) private(i, tmp, fase)//Usando default fuerzo a decidir el scope de TODAS las variables  
for (fase=0; fase<5;fase++)  
{  
 if(fase % 2 == 0)  
 #pragma omp for  
 for (i=1;i<5;i+=2)  
 {  
 if(v2[i-1]>v2[i])  
 {  
 tmp = v2[i-1];  
 v2[i-1] = v2[i];  
 v2[i] = tmp;  
 }  
 }  
 else  
 #pragma omp for  
 for(i=1;i<5-1;i+=2)  
 {  
 if(v2[i]>v2[i+1])  
 {  
 tmp=v2[i+1];  
 v2[i+1] = v2[i];  
 v2[i]= tmp;  
 }  
 }  
 }  
 cout<<"odenamiento de burbujeo paralelo ";  
#pragma omp for  
 for(i=0;i<5;i++){  
 cout<<" ";  
 cout<<v2[i];  
 }  
 return 0;  
 }

1. Ordenamiento Cocktail Sort
2. Performance para el peor caso: О(*n*2)
3. Performance para el mejor caso: O(*n*)
4. Complejidad: O(n²)

#include <iostream>  
using namespace std;  
 int main()  
 {  
 int a;  
 int cambio;  
 int v2[] = {3,2,1,5,4};  
 int t;  
 do  
 {  
 cambio = 0;  
 for(a=5-1; a>0; --a)  
 {  
 if(v2[a-1] > v2[a])  
 {  
 t = v2[a-1];  
 v2[a-1] = v2[a];  
 v2[a] = t;  
 cambio = 1;  
 }  
 }  
 for(a=1; a < 5; ++a)  
 {  
 if(v2[a-1] > v2[a])  
 {  
 t = v2[a-1];  
 v2[a-1] = v2[a];  
 v2[a] = t;  
 cambio = 1;  
 }  
 }  
 }while(cambio);  
 cout<<"ordenamiento de Cocktail Sort ";  
  
 for(a=0;a<5;a++){  
 cout<<" ";  
 cout<<v2[a];  
 }  
 return 0;  
 }

2.2 ordenamiento en paralelo Cocktail Sort

#include <iostream>

#include <omp.h>  
 using namespace std;  
 int main()  
 {  
 int a,cambio;  
 int v2[] = {3,2,1,5,4}; //vector desordenado  
 int t=0;  
 #pragma omp parallel default(none) shared(v2) private(a, t, cambio)  
 do  
 {  
 cambio = 0;  
 #pragma omp for  
 for(a=5-1; a>0; --a)  
 {  
 if(v2[a-1] > v2[a])  
 {  
 t = v2[a-1];  
 v2[a-1] = v2[a];  
 v2[a] = t;  
 cambio = 1;  
 }  
 }  
 #pragma omp for  
 for(a=1; a < 5; ++a)  
 {  
 if(v2[a-1] > v2[a])  
 {  
 t = v2[a-1];  
 v2[a-1] = v2[a];  
 v2[a] = t;  
 cambio = 1;  
 }  
 }  
 }while(cambio);  
 cout<<"ordenamiento Paralelo de Cocktail Sort ";  
 #pragma omp for  
 for(a=0;a<5;a++){  
 cout<<" ";  
 cout<<v2[a];  
 }  
 return 0;  
 }

3 ordenamiento Shell Sort

Performance para el peor caso: O(*n*log2 2*n*)

Performance para el mejor caso: O(*n* log *n*)

Complejidad: О(n)

* 1. Shell Sort lineal o secuencial

#include <iostream>  
using namespace std;  
 int main()  
 {  
 int i, j, incrmnto=0, temp;  
 int v2[] = {3,2,1,5,4}; //vector desordenado =NARRAY = 5  
 incrmnto = 5/2;  
 while (incrmnto > 0)  
 {  
 for (i=incrmnto; i < 5; i++)  
 {  
 j = i;  
 temp = v2[i];  
 while ((j >= incrmnto) && (v2[j-incrmnto] > temp))  
 {  
 v2[j] = v2[j - incrmnto];  
 j = j - incrmnto;  
 }  
 v2[j] = temp;  
 }  
 incrmnto /= 2;  
 }  
 cout<<"ordenamiento secuencial Shell Sort ";  
 for(i=0;i<5;i++){  
 cout<<" ";  
 cout<<v2[i];  
 }  
 return 0;  
 }

* 1. Shell Sort Paralelo OMP

#include <iostream>  
#include <omp.h>  
 using namespace std;  
 int main()  
 {  
 int i=0, j=0, incrmnto=0, temp=0;  
 int v2[] = {3,2,1,5,4}; //vector desordenado NARRAY = 5  
 incrmnto = 5/2;  
 #pragma omp parallel default(none) firstprivate(incrmnto) shared(v2) private(i, j, temp)//Con firstprivate mantengo el valor inicializado anteriormente de incrmnto  
 {  
 while (incrmnto > 0)  
 {  
 #pragma omp for  
 for (i=incrmnto; i < 5; i++)  
 {  
 j = i;  
 temp = v2[i];  
 while ((j >= incrmnto) && (v2[j-incrmnto] > temp))  
 {  
 v2[j] = v2[j - incrmnto];  
 j = j - incrmnto;  
 }  
 v2[j] = temp;  
 }  
 incrmnto /= 2;  
 }  
 }  
 cout<<"ordenamiento Paralelo de Shell Sort ";  
 #pragma omp for  
 for(i=0;i<5;i++){  
 cout<<" ";  
 cout<<v2[i];  
 }  
 return 0;  
 }

1. Ordenamiento Insertion Sort

Performance para el peor caso: О(*n*2) comparaciones, swaps

Performance para el mejor caso: O(*n*) comparaciones, O(*1*) swaps

Complejidad: O(n²)

* 1. Ordenamiento secuencial insetion sort

#include <iostream>  
 using namespace std;  
 int main()  
 {  
 int i, j, aux;  
 int v2[] = {3,2,1,5,4}; //vector desordenado NARRAY = 5  
  
 for (i=1; i<5; i++)  
 {  
 aux = v2[i];  
 j = i;  
 while ((j>0) && (v2[j-1] > aux))  
 {  
 v2[j] = v2[j-1];  
 j = j - 1;  
 }  
 v2[j] = aux;  
 }  
  
 cout<<"Ordenamiento Secuencial de Insertion Sort ";  
 for(i=0;i<5;i++){  
 cout<<" ";  
 cout<<v2[i];  
 }  
 return 0;  
 }

* 1. Ordenamiento paralelo insertion sort

#include <iostream>  
#include <omp.h>  
 using namespace std;  
 int main()  
 {  
 int i, j, aux;  
 int v2[] = {3,2,1,5,4}; //vector desordenado NARRAY = 5  
 #pragma omp parallel for default(none) shared(v2) private(i, j, aux)  
 for (i=1; i<5; i++)  
 {  
 aux = v2[i];  
 j = i;  
 while ((j>0) && (v2[j-1] > aux))  
 {  
 v2[j] = v2[j-1];  
 j = j - 1;  
 }  
 v2[j] = aux;  
 }  
 cout<<"ordenamiento Paralelo de Insertion sort ";  
 for(i=0;i<5;i++){  
 cout<<" ";  
 cout<<v2[i];  
 }  
 return 0;  
 }

1. CONCLUSION

Un que no parece tan relevante los algoritmos de ordenamiento, pero ala hora tener muchos datos se nos hace muy útil y además que asu ve necesitamos de manera instantánea entonces en estos casos podemos emplear OMP para poder paralelizar los algoritmos de ordenamiento. Como se observo reducen cierta cantidad de tiempo según el ordenador y el procesador de ellos

1. **Bibliografía**

TIM Mattsonhttps [OpenMP].(2013,diciembre 6) recuperado de ://www.youtube.com/watch?v=sNmGXNBzofM&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG&index=27

<https://openwebinars.net/blog/que-es-scope-en-c/>

<https://openwebinars.net/blog/como-depurar-c-visual-studio/>

<https://www.geeksforgeeks.org/understanding-register-keyword/>

<https://www.youtube.com/watch?v=ZD9yICbSyBQ>

<https://en.wikipedia.org/wiki/Bubble_sort>

<https://en.wikipedia.org/wiki/Cocktail_shaker_sort>

<https://en.wikipedia.org/wiki/Shellsort>

<https://en.wikipedia.org/wiki/Insertion_sort>