PRÁCTICA 13

ALGORITMO PARALELOS PARTE II

## Alumno: Robles Martínez Héctor

## Número de Cuenta UNAM: 317356548

## Semestre: Tercer semestre de Ingeniería en Computación (2016)

## Semestre Actual: 2021-1

## Profesor: Jorge A. Solano G.

Objetivo:

Utilizar algunos patrones paralelos para resolver algunos problemas de forma paralela y con ello adquirir experiencia en el desarrollo de programas multihilo en sistemas multiprocesador de memoria compartida.

Instrucciones:

• Realizar programas que resuelvan algunos problemas y algoritmos de forma paralela

utilizando distintas directivas de OpenMP.

Resultados obtenidos:

Completar el siguiente código para que se pueda ejecutar con la cláusula nowait utilizando

OpenMP y lenguaje C.

#include <stdio.h>

#include <unistd.h>

#include <omp.h>

#define TAM 12

#define THN 12

int work1(int id){

sleep(id);

return (int)(id%10);

}

int work2(int id, int\* A){

sleep(id);

int res = 0;

for(int i = 0; i<= id; i++){

res += A[i];

}

return res;

}

int work3(int\* C, int id){

return (int)(id);

}

int work4(int id){

sleep(id);

return 0;

}

int main(){

int A[THN + 1], B[TAM + 1], C[TAM + 1], id;

omp\_set\_num\_threads(THN);

#pragma omp parallel shared(A, B, C), private(id)

{

int i;

id = omp\_get\_thread\_num();

A[id] = work1(id);

printf("Thread %d finish work1\n", id);

#pragma omp barrier

#pragma omp for

for(i = 0; i<TAM; i++){

C[i] = work2(i, A);

}

printf("Thread %d finish work2\n", id);

#pragma omp for nowait

for(i = 0; i < TAM; i++){

B[i] = work3(C, i);

}

printf("Thread %d finish work3\n", id);

A[id] = work4(id);

printf("Thread %d finish work4\n", id);

}

return 0;

}

Salida:

Thread 0 finish work1

Thread 1 finish work1

Thread 2 finish work1

Thread 3 finish work1

Thread 4 finish work1

Thread 5 finish work1

Thread 6 finish work1

Thread 7 finish work1

Thread 8 finish work1

Thread 9 finish work1

Thread 10 finish work1

Thread 11 finish work1

Thread 0 finish work2

Thread 0 finish work3

Thread 11 finish work2

Thread 8 finish work2

Thread 2 finish work2

Thread 5 finish work2

Thread 5 finish work3

Thread 6 finish work2

Thread 6 finish work3

Thread 7 finish work2

Thread 7 finish work3

Thread 3 finish work2

Thread 3 finish work3

Thread 11 finish work3

Thread 9 finish work2

Thread 9 finish work3

Thread 0 finish work4

Thread 1 finish work2

Thread 1 finish work3

Thread 8 finish work3

Thread 2 finish work3

Thread 4 finish work2

Thread 4 finish work3

Thread 10 finish work2

Thread 10 finish work3

Thread 1 finish work4

Thread 2 finish work4

Thread 3 finish work4

Thread 4 finish work4

Thread 5 finish work4

Thread 6 finish work4

Thread 7 finish work4

Thread 8 finish work4

Thread 9 finish work4

Thread 10 finish work4

Thread 11 finish work4

Completar el siguiente código para que se pueda ejecutar con el constructor master utilizando OpenMP y lenguaje C:

#include <stdio.h>

#include <omp.h>

#include <unistd.h>

#define NUM\_THREADS *12*

int main(){

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

int id;

#pragma omp parallel

{

#pragma omp master

{

id = omp\_get\_thread\_num();

printf("Master block thread %d \n", id);

}

id = omp\_get\_thread\_num();

printf("Parallel block thread %d\n", id);

}

}

Salida:

Parallel block thread 10

Parallel block thread 11

Master block thread 0

Parallel block thread 0

Parallel block thread 6

Parallel block thread 7

Parallel block thread 9

Parallel block thread 8

Parallel block thread 1

Parallel block thread 4

Parallel block thread 3

Parallel block thread 5

Parallel block thread 2

Completar el siguiente código para que se pueda ejecutar con el constructor single utilizando OpenMP y lenguaje C.:

#include <stdio.h>

#include <omp.h>

#include <unistd.h>

#define NUM\_THREADS 24

int main(){

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

int id;

#pragma omp parallel

{

#pragma omp single

{

id = omp\_get\_thread\_num();

printf("Single block thread %d \n", id);

}

id = omp\_get\_thread\_num();

printf("Parallel block thread %d\n", id);

}

}

Salida:

Completar el siguiente código para que se pueda ejecutar con el constructor sections utilizando OpenMP y lenguaje C.:

Salida:

Single block thread 7

Parallel block thread 2

Parallel block thread 5

Parallel block thread 1

Parallel block thread 11

Parallel block thread 8

Parallel block thread 3

Parallel block thread 7

Parallel block thread 4

Parallel block thread 9

Parallel block thread 10

Parallel block thread 0

Parallel block thread 6

Completar el siguiente código para que se pueda ejecutar con la función lock utilizando OpenMP y lenguaje C.:

#include <stdio.h>

#include <omp.h>

#include <stdlib.h>

#include <unistd.h>

#define NUM\_VALUES 20

int NUM\_BUCKETS = 0;

int takeNumber(){

return rand()%NUM\_BUCKETS;

}

int main(){

int i;

omp\_set\_dynamic(0);

NUM\_BUCKETS = omp\_get\_num\_procs();

omp\_set\_num\_threads(NUM\_BUCKETS);

printf("Buckets: %d\n", NUM\_BUCKETS);

omp\_lock\_t hist\_locks[NUM\_BUCKETS];

int hist[NUM\_BUCKETS];

#pragma omp parallel for

for(i = 0; i < NUM\_BUCKETS; i++){

omp\_init\_lock(&hist\_locks[i]);

hist[i] = 0;

}

#pragma omp parallel for

for(i = 0; i < NUM\_VALUES; i++){

int id = omp\_get\_thread\_num();

printf("Thread: %d\n", id);

int val = takeNumber();

omp\_set\_lock(&hist\_locks[id]);

hist[val] ++;

omp\_unset\_lock(&hist\_locks[id]);

}

for(i = 0; i < NUM\_BUCKETS; i++){

printf("hist[%d] = %d\n", i, hist[i]);

omp\_destroy\_lock(&hist\_locks[i]);

}

return 0;

}

Salida:

Buckets: 12

Thread: 6

Thread: 6

Thread: 2

Thread: 2

Thread: 5

Thread: 5

Thread: 10

Thread: 3

Thread: 3

Thread: 4

Thread: 4

Thread: 1

Thread: 1

Thread: 7

Thread: 7

Thread: 8

Thread: 0

Thread: 0

Thread: 11

Thread: 9

hist[0] = 2

hist[1] = 1

hist[2] = 2

hist[3] = 0

hist[4] = 2

hist[5] = 1

hist[6] = 1

hist[7] = 5

hist[8] = 0

hist[9] = 2

hist[10] = 3

hist[11] = 1

Completar el siguiente código para que se pueda ejecutar el deadlock con la función lock utilizando OpenMP y lenguaje C.:

#include <stdio.h>

#include <omp.h>

#define N 100

int main(){

int A[N], B[N], i, threads;

omp\_lock\_t locka, lockb;

#pragma omp sections nowait

{

printf("Section one");

#pragma omp section

{

omp\_set\_lock(&locka);

printf("init a\n");

for(i = 0; i < N; i++){

A[i] = i;

}

omp\_set\_lock(&lockb);

printf("init b\n");

for(i = 0; i < N; i++){

B[i] = N - A[i];

}

omp\_unset\_lock(&lockb);

omp\_unset\_lock(&locka);

}

#pragma omp section

{

printf("Section two\n");

omp\_set\_lock(&lockb);

printf("Modify b\n");

for(i = 0; i < N; i++){

B[i] = N-i;

}

omp\_set\_lock(&locka);

printf("Modify a\n");

for(i = 0; i < N; i++){

A[i] = B[i] + i;

}

omp\_unset\_lock(&locka);

omp\_unset\_lock(&lockb);

}

}

return 0;

}

Salida:

hector@hpOmen:~/EDA2/openmp/bloqueo/deadLock$ ./a.out

Section oneinit a

^C

hector@hpOmen:~/EDA2/openmp/bloqueo/deadLock$ ./a.out

Section oneinit a

init b

Section two

Modify b

Modify a

hector@hpOmen:~/EDA2/openmp/bloqueo/deadLock$

Completar el siguiente código para que se pueda ejecutar con el siguiente código utilizando OpenMP y lenguaje C.:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 10

double\* A;

void fillRand(){

for(int i = 0; i < N; i ++){

A[i] = rand()%N;

}

}

double sumArray(){

double sum;

for(int i = 0; i < N; i++){

sum += A[i];

}

return sum;

}

int main(){

double sum, runtime;

int flag = 0;

A = (double \*) malloc(N\*sizeof(double));

runtime = omp\_get\_wtime();

fillRand();

sum = sumArray();

runtime = omp\_get\_wtime() - runtime;

printf("Sum = %lf\nRuntime = %lf\n", sum, runtime);

return 0;

}

Salida:

Sum = 47.000000

Runtime = 0.000001

Completar el siguiente código para que se pueda ejecutar con el constructor flush utilizando OpenMP y lenguaje C.:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 10

double\* A;

void fillRand(){

for(int i = 0; i < N; i ++){

A[i] = rand()%N;

}

}

double sumArray(){

double sum;

for(int i = 0; i < N; i++){

sum += A[i];

}

return sum;

}

int main(){

double sum, runtime;

int flag = 0;

A = (double \*) malloc(N\*sizeof(double));

runtime = omp\_get\_wtime();

#pragma omp parallel sections num\_threads(2)

{

#pragma omp section

{

fillRand();

#pragma omp flush //flush the array (espera a que termine la función anterior)

flag = 1;

#pragma omp flush(flag)

}

#pragma omp section

{

#pragma omp flush(flag)

while(flag == 0){

#pragma omp flush(flag)

}

#pragma omp flush

sum = sumArray();

}

}

runtime = omp\_get\_wtime() - runtime;

printf("Sum = %lf\nRuntime = %lf\n", sum, runtime);

return 0;

}

Salida:

hector@hpOmen:~/EDA2/openmp/flush/parallel/first$ ./a.out

Sum = -23682837551127393809865858783078329374010771802161561633568119319355010006331105494553646449272930877395692756048210303000502311080172912178476937640594467739079228260352.000000

Runtime = 0.000155

hector@hpOmen:~/EDA2/openmp/flush/parallel/first$ ./a.out

Sum = 47.000000

Runtime = 0.000178

hector@hpOmen:~/EDA2/openmp/flush/parallel/first$ ./a.out

Sum = -33498142646058562145872385146480597822676086422889784585739463271216270719792871238404656269266886819422532534272.000000

Runtime = 0.000160

hector@hpOmen:~/EDA2/openmp/flush/parallel/first$

Completar el siguiente código para que se pueda ejecutar con el constructor flush utilizando OpenMP y lenguaje C.:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 10

double\* A;

void fillRand(){

for(int i = 0; i < N; i ++){

A[i] = rand()%N;

}

}

double sumArray(){

double sum;

for(int i = 0; i < N; i++){

sum += A[i];

}

return sum;

}

int main(){

double sum, runtime;

int flag = 0, tmp\_flag = 0;

A = (double \*) malloc(N\*sizeof(double));

runtime = omp\_get\_wtime();

#pragma omp parallel sections num\_threads(2)

{

#pragma omp section

{

fillRand();

#pragma omp flush //flush the array (espera a que termine la función anterior)

#pragma omp atomic write

flag = 1;

#pragma omp flush(flag)

}

#pragma omp section

{

#pragma omp flush(flag)

while(1){

#pragma omp flush(flag)

#pragma omp atomic read

tmp\_flag = flag;

if(tmp\_flag == 1){

break;

}

}

#pragma omp flush

sum = sumArray();

}

}

runtime = omp\_get\_wtime() - runtime;

printf("Sum = %lf\nRuntime = %lf\n", sum, runtime);

return 0;

}

Salida:

hector@hpOmen:~/EDA2/openmp/flush/parallel/second$ ./a.out

Sum = 1021454294439232569083802157056.000000

Runtime = 0.000163

hector@hpOmen:~/EDA2/openmp/flush/parallel/second$ ./a.out

Sum = -1153304454566398046921689133130637133768219776627495455169653499201867018959040714916185712398597432795180774660583847721621065966393753600.000000

Runtime = 0.000189

hector@hpOmen:~/EDA2/openmp/flush/parallel/second$ ./a.out

Sum = 47.000000

Runtime = 0.000154

hector@hpOmen:~/EDA2/openmp/flush/parallel/second$ ./a.out

Sum = -941483799030303349391535384932393758761260053154340896413308341328988965798961393922044986590000868796995795831509092858881117508059195149339880204082038034745274644839556174919046791168.000000

Runtime = 0.000154

Completar el siguiente código para que se pueda ejecutar con la cláusula reduction

utilizando OpenMP y lenguaje C.:

#include <stdio.h>

#include <omp.h>

#define MAX 5

int main(){

double ave = 0.0, A[MAX];

int i;

for(i = 0; i < MAX; i++){

A[i] = i+1.0;

}

#pragma omp parallel for reduction(+:ave)

for(i = 0; i < MAX; i++){

ave += A[i];

}

ave /= MAX;

printf("%f\n", ave);

return 0;

}

Salida:

3.000000

Conclusiones:

Esta práctica me sirvió para comprender cómo se pueden implementar los hilos y procesos paralelos en diferentes problemas y escenarios en la resolución de problemas, podiendo resolver los conflictos que se generen debido a la inconsistencia de datos.