Rubén Vera - 241456
Sara Soriano - 240007

REPORT LAB1

Exercici 1. Escriu un codi en C que implementi el producte escalat entre vectors d'*ints*.

Per dur a terme aquest exercici hem implementat les següents línies de codi:

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
    int size, ex, nthreads;

    sscanf(argv[3], "%d", &size);
    sscanf(argv[2], "%d", &nthreads);
    sscanf(argv[1], "%d", &ex);

    int total = 0;
    int* a = malloc(sizeof(int)*size);
    int* b = malloc(sizeof(int)*size);
    srand(1);
    for(int i = 0; i < size; i++){
        a[i] = (int) (rand() % size - size/2);
        b[i] = (int) (rand() % size - size/2);
    }
    if(ex == 1){
        printf("Exercici 1\n");
        double start = omp_get_wtime();
        for(int i=0; i < size; i++){
            total+=a[i]*b[i];
        }
        double end = omp_get_wtime();
        double time = end-start;
        printf("%.4e\t%i\n", time, total);
    }
```

Primer de tot hem inclós els diccionaris de C necessaris per les instruccions que fariem a continuació i hem assignat cada argument d'entrada a una variable per tal de poder treballar amb més facilitat. A continuació, simplement hem generat els vectors a i b, com se'ns demanava en l'enunciat de la pràctica, i hem inicialitzat el *seed* amb la instrucció *srand(1)*.
Com aquest codi l'anàvem a reutilitzar pels exercicis 2 i 3, hem establert un if per tal de que només s'executi aquell codi quan estiguem a l'exercici 1. Dins d'ell, simplement hem iniciat un contador i hem calculat el producte escalar, com sempre hem fet. I per acabar hem tancat el contador i es mostra per pantalla el temps que s'ha trigat en fer el producte i el resultat d'aquest.

Rubén Vera - 241456
Sara Soriano - 240007

Exercici 2.

- **Paral·lelitzeu el codi mitjançant OpenMP.**

  Per realitzar aquesta part de codi, hem introduït una altre condició que controlés quan ens trobem en l'exercici 2. Allà com es mostra en les següents línies de codi hem creat els threads que se'ns demanen (el nombre de threads que s'entra com a segon paràmetre en el programa) i com en l'exercici anterior hem iniciat un contador. El següent pas ha sigut paral·lelitzar el producte escalar de manera que el threads actuïn paral·lelament amb l'objectiu de que cadascun realitzi uns productes escalars. I per acabar amb aquesta part de codi simplement hem hagut de tancar el contador i mostrar per pantalla els mateixos resultats que en l'exercici anterior.

```
else if(ex == 2){
    printf("Exercici 2, nthreads %d\n", nthreads);
    omp_set_num_threads(nthreads);
    int i;
    double start = omp_get_wtime();
    #pragma omp parallel for reduction(+:total)
    for(i = 0; i < size; i++){
        total+=a[i]*b[i];
    }
    double end = omp_get_wtime();
    double time = end-start;
    printf("%.4e\t%i\n", time, total);
}
```

- **Envieu un treball executant el codi amb una mida vectorial de 100 milions d'enters i obteniu resultats per a 1, 2, 4 i 8 threads. Com s'escala? Traça l'acceleració forta.**

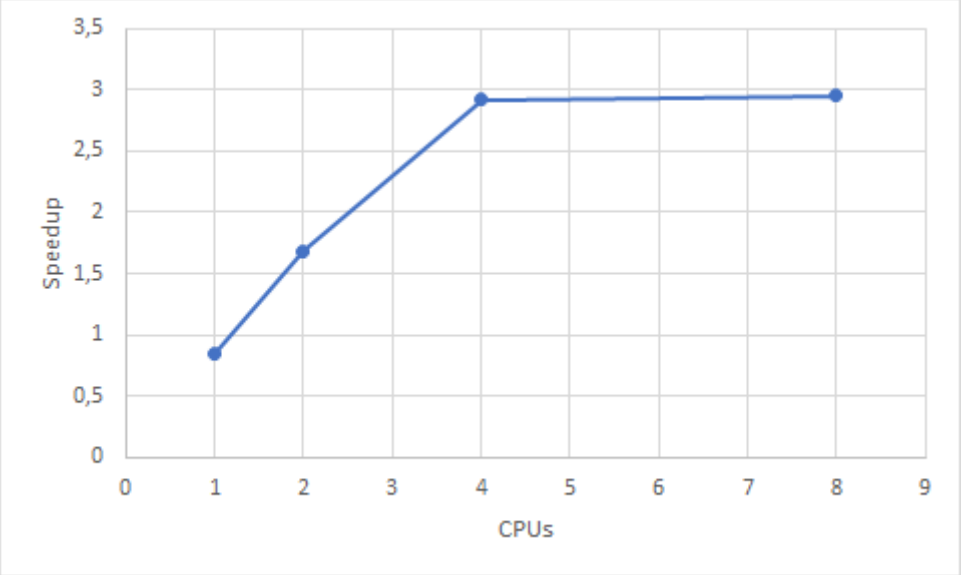
```
#!/bin/bash
#SBATCH --job-name=test
#SBATCH --output=out_dotp.out
#SBATCH --error=out_dotp.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --time=00:00:10

source /etc/profile.d/z00-global-profile.sh

module load GCC/10.2.0

make clean >> out_dotp.err && make >> out_dotp.err || exit 1

./a.out 1 1 100000000
./a.out 2 1 100000000
./a.out 2 2 100000000
./a.out 2 4 100000000
./a.out 2 8 100000000
```

Com es pot observar de l'anterior imatge i en les últimes 4 línies rebem els resultats dels treballs d'executar el codi amb mida vectorial de 100 milions d'enters i per 1, 2, 4 i 8 threads.

Com es pot veure en la imatge següent, l'escala es logarítmica descendent, és a dir, a mesura que augmenta el número de cores, disminueix el temps d'execució.

```
Exercici 2, nthreads 1
6.2915e-02      -1525028084
Exercici 2, nthreads 2
3.1998e-02      -1525028084
Exercici 2, nthreads 4
2.1702e-02      -1525028084
Exercici 2, nthreads 8
2.1289e-02      -1525028084
```

Veient els resultats amb el gràfic speedup/CPUs, l'escala del gràfic es logaritmica, la qual cosa succeeix perquè els recursos estan doblats. En el següent gràfic s'observa l'*strong speed*.



- **Utilitzant el temps dedicat a la funció de producte puntual, calculeu el rendiment en GBytes/s i GFLOP/s per a cada configuració.**

```
Exercici 2 1
7.4274e-01      -1362393297
Exercici 2 2
3.7611e-01      -1362393297
Exercici 2 4
2.1747e-01      -1362393297
Exercici 2 8
2.1526e-01      -1362393297
```

Rubén Vera - 241456
Sara Soriano - 240007

	1	2	4	8
GBytes/s	$\frac{0.1Gbytes*4}{7.4274e^{-01}s} = 0.54$	$\frac{0.1Gbytes*4}{3.7611e^{-01}s} = 1.06$	$\frac{0.1Gbytes*4}{2.1747e^{-01}s} = 1.84$	$\frac{0.1Gbytes*4}{2.1526e^{-01}s} = 1.86$
GFLOPS/s	$\frac{0.1Gbytes*2}{7.4274e^{-01}s} = 0.27$	$\frac{0.1Gbytes*2}{3.7611e^{-01}s} = 0.53$	$\frac{0.1Gbytes*2}{2.1747e^{-01}s} = 0.92$	$\frac{0.1Gbytes*2}{2.1526e^{-01}s} = 0.93$

Exercici 3.

- **Utilitzeu la vectorització per paral·lelitzar encara més. Si el bucle està vectoritzat correctament, la compilació mostrarà el missatge: *dotp.c:xx:yy: optimized: bucle vectoritzat amb vectors de 32 bytes.***

  Per tal de vectoritzar, com hem vist a teoria, utilitzem SIMD. L'única modificació que hem hagut de fer en comparació amb el exercici 2 és el pragma on hem afegit simd. Que això provoca que el codi es vectoritzi i, per tant, és paral·lelitzi encara més.

```
else{
    printf("Exercici 3, nthreads %d\n", nthreads);
    double start = omp_get_wtime();
    #pragma omp parallel for simd reduction(+:total)
    for(int i = 0; i < size; i++){
        total+=a[i]*b[i];
    }
    double end = omp_get_wtime();
    double time = end - start;
    printf("%.4e\t%i\n", time, total);
}
```

  Un cop compilat el codi i executat veiem en el codi out_dotp.err el següent missatge que com ens diu l'enunciat ens mostra que s'ha vectoritzat correctament.

```
dotp.c:48:14: optimized: loop vectorized using 32 byte vectors
```

- **Envieu un treball executant el codi amb una mida vectorial de 100M de nombres enters i obteniu resultats per a 1, 2, 4 i 8 nuclis amb l'opció de vectorització dins de la regió paral·lela. Com s'escala? Afegiu una taula al vostre informe comparant els valors amb la paral·lelització simple.**

```
Exercici 2, nthreads 1
6.2915e-02      -1525028084
Exercici 2, nthreads 2
3.1998e-02      -1525028084
Exercici 2, nthreads 4
2.1702e-02      -1525028084
Exercici 2, nthreads 8
2.1289e-02      -1525028084
Exercici 3, nthreads 1
2.0794e-02      -1525028084
Exercici 3, nthreads 2
2.0882e-02      -1525028084
Exercici 3, nthreads 4
2.1183e-02      -1525028084
Exercici 3, nthreads 8
2.0804e-02      -1525028084
```

	1	**2**	**4**	**8**
Simple	0.062915	0.031998	0.021702	0.021289
Vectoritzat	0.020794	0.020882	0.021183	0.020804

Com veiem en la taula, en el cas de fer l'execució amb els nuclis vectoritzats l'escala es lineal i no logarítmica descendent com en el cas de nucli amb opció simple.

- **Canvieu la marca -O2 al Makefile per -O3. Envieu un treball executant el codi amb una mida vectorial de 100M de nombres enters i obteniu resultats per a 1, 2, 4 i 8 nuclis amb les opcions de l'exercici 2 i l'exercici 3. Compara els resultats. Què fa -O3?**

-O2 o -O3 és el nivell d'optimització del codi. Amb -O3, GCC es vectoritza, per tant els nuclis estan vectoritzats sense necessitat de *#pragma omp simd* i com es pot veure en les següents imatges, ara l'escala de temps de ambdós exercicis es lineal. La primera imatge que es mostra a continuació correspon a la marca -O2 i la segona a la marca -O3.

```
Exercici 2, nthreads 1
6.2915e-02      -1525028084
Exercici 2, nthreads 2
3.1998e-02      -1525028084
Exercici 2, nthreads 4
2.1702e-02      -1525028084
Exercici 2, nthreads 8
2.1289e-02      -1525028084
Exercici 3, nthreads 1
2.0794e-02      -1525028084
Exercici 3, nthreads 2
2.0882e-02      -1525028084
Exercici 3, nthreads 4
2.1183e-02      -1525028084
Exercici 3, nthreads 8
2.0804e-02      -1525028084
```

```
Exercici 2, nthreads 1
3.1539e-02      -1525028084
Exercici 2, nthreads 2
1.6373e-02      -1525028084
Exercici 2, nthreads 4
1.6070e-02      -1525028084
Exercici 2, nthreads 8
2.1420e-02      -1525028084
Exercici 3, nthreads 1
3.0560e-02      -1525028084
Exercici 3, nthreads 2
1.8354e-02      -1525028084
Exercici 3, nthreads 4
4.1811e-02      -1525028084
Exercici 3, nthreads 8
2.0323e-02      -1525028084
```

Rubén Vera - 241456
Sara Soriano - 240007

Exercici 4.

- **Utilitzeu tasques per paral·lelitzar les trucades recursives a Quicksort. S'ha de crear una regió paral·lela i un sol fil ha de crear tasques.**

  Com vam veure a teoria per paral·lelitzar a partir de tasques hem de fer servir *#pragma omp task*, ja que generarà una tasca per realitzar el codi que estigui entre els claudàtors. A més a més, com volem que això només ho faci un fill cal que abans de la primera crida és posi *#pragma omp single*, i abans d'aquest *#pragma omp parallel* per paral·lelitzar el codi.

```
void Quicksort(int *a, int lo, int hi){
    if ( lo < hi ) {
        int X =10;
        int p = partition(a, lo, hi);
        #pragma omp task
        {if(hi-lo<=(X)){(void) Quicksort(a, lo, p - 1);}} // Left branch
        #pragma omp task
        {if(hi-lo<=(X)){(void) Quicksort(a, p + 1, hi);}} // Right bran
    }
}

int main(int argc, char *argv[])
{
    srand(1);

    int size;
    int counter = 0;

    long long elp = 0;

    if (argc < 1 || sscanf(argv[1], "%i", &size)!=1) {printf("\n\nPlease add size as argument.\n\n")
; return -1;}

    int *a = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {a[i] = rand() % size;}

    start_timer();
    #pragma omp parallel
    {
        #pragma omp single
        {
            Quicksort(a, 0, size - 1);
        }
    }
```

- **Envieu un treball executant el codi amb una mida vectorial d'1M dobles i obteniu resultats per a 1, 2, 4, 8 i 16 nuclis (podeu utilitzar el fitxer de treball de mostra dotp sort.cmd com a base per als fitxers de treball). Com s'escala? Traça l'acceleració.**

  Els resultats que hem obtingut a partir de 1, 2, 4, 8 i 16 nuclis són els que es mostren en la taula següent:

1	2	4	8	16
0.092401	0.048154	0.029077	0.099977	0.081715

  A partir de les dades que hem obtingut en aquest estudi, veiem que podria escalar-se com una paràbola. Però com són operacions poc costoses i que prenen molt de temps depèn de l'execució no segueix una escala de manera consistent.

- **Envieu un treball executant el codi amb una mida vectorial de 100 M dobles i obteniu resultats per a 1, 2, 4, 8 i 16 nuclis (podeu utilitzar el fitxer de treball de mostra dotp sort.cmd com a base per als fitxers de treball). S'escala millor o pitjor que abans? Per què?**

  A partir de les dades que hem obtingut en aquest estudi, aquest cop amb 100M dobles, veiem que estàvem en el correcte i al ser l'estudi sobre 1M no podíem estar segurs de l'escala. Però amb aquest estudi podem confirmar que l'escala és logarítmica descendent, és a dir, a més cores menys temps, però no linealment.

1	2	4	8	16
11.773823	5.789451	4.04133	2.302720	2.340548

Exercici 5.

- **Afegiu la clàusula *if (hi − lo >= (X))* als pragmas de les trucades recursives a Quicksort i repetiu les proves dels exercicis 4.2 i 4.3. Proveu diferents valors per a (X) (5, 10 i 1000) i diferents mides per al problema. Què observes? Què fa? Afegiu una taula al vostre informe amb els diferents valors que heu obtingut.**

4.2. (1M dobles)

	1	2	4	8	16
5	0.089268	0.044894	0.025848	0.027195	0.046358
10	0.081011	0.041637	0.024350	0.027772	0.036850
1000	0.043146	0.023219	0.013870	0.012229	0.013121

4.3. (100M dobles)

	1	2	4	8	16
5	11.169139	5.685290	2.955042	2.070909	1.984356
10	10.285774	5.210282	2.844347	1.896798	1.725336
1000	6.538761	3.359081	1.847713	1.280945	1.153633

Com es pot veure en les taules anteriors, en el cas d'usar 100M de dobles, a mesura que s'incrementa la X, els temps d'execució es redueixen, i a mesura que utilitzem més cores per tasca el temps es veu reduït. També es pot veure com la diferència entre utilitzar 1 o 2 cores és molt major a la diferència entre utilitzar 8 o 16. Això es deu al fet que el temps d'execució és en escala logarítmica i que arriba un punt que per molt que augmentin els cores, la diferència de temps no serà tan notòria. En canvi, en el cas d'1M de dobles, tot i que es veu que l'escala hauria de ser igual, hi ha alguns casos on no es compleix pel fet que el temps és tan petit que a vegades amb 8 cores és capaç de fer-ho més ràpid que en 16.

Exercici 6.

- **Utilitzeu *gprof* per obtenir un perfil del codi. Quina és la funció que requereix més temps?**

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
99.86	7.27	7.27	380575	0.02	0.02	Fitness
0.14	7.28	0.01				main
0.00	7.28	0.00	7798	0.00	0.00	Tournament
0.00	7.28	0.00	7300	0.00	0.98	Mutation
0.00	7.28	0.00	3650	0.00	0.00	OnePointCrossover
0.00	7.28	0.00	74	0.00	1.91	population_fitness
0.00	7.28	0.00	73	0.00	0.00	copy_pop
0.00	7.28	0.00	1	0.00	0.00	create_population

Com es pot veure a la imatge anterior el % time es el percentatge del temps total d'execució que cada funció ha utilitzat. Com s'observa, el 99.86% del temps ha estat utilitzat per la funció Fitness, per aquest motiu Fitness és la funció que requereix més temps.

- **Paral·lelitzeu la funció que consumeix més temps i executeu el codi sol·licitant 4 nuclis. El nou codi és més ràpid?**

Per fer el codi paral·lel i per tal de que funcioni correctament, en la funció Fitness hem introduït la següent línia de codi per tal de "bloquejar" els dos bucles fors: *#pragma cmd parallel for private(k) reduction(+:attack)*.
A la següent imatge, que mostra el resultat del fitxer out_nqueens.out, es veu que el total time es de 3.49 segons quan en l'apartat anterior previ a paralelitzar el temps era de 7.3 segons.

```
[ul86665@ip-10-49-0-109 3_nqueens]$ cat out_nqueens.out

N-Queens problem. Number of queens = 100

Problem Solved: Fitness = 0. Iteration: 73. Total Time: 3.491150
```