Adrià Muro Gómez (1665191)
David Morillo Massagué (1666540)

Exercicis Pràctics OpenMP (Part II)

L'objectiu principal d'aquests exercicis és que experimenteu amb les capacitats d'OpenMP fent servir casos simples, facilitant d'aquesta manera la transició entre els continguts discutits en la class de teoria i la seva aplicació al cas pràctic (més complex) treballat en el laboratori.

El plantejament dels exercicis i la mecànica de treball per resoldre'ls consisteixen en:
1) Per cada apartat, es proporcionen un conjunt de fragments de codi que caldrà executar i analitzar (els programes corresponents els trobareu a /home/alumnos/pp/alumnos/Avaluats-problemes/OpenMP/sessio2/, copieu els arxius al vostre compte per fer els problemes).
2) Per cada apartat, es fa un conjunt de preguntes sobre cada fragment de codi presentat. Heu de respondre cada pregunta, **justificant sempre la vostra resposta**. En alguns casos la justificació serà molt curta, en altres més complexa i, en altres, potser consistirà en un nou fragment de codi.
3) La solució als exercicis s'inclourà en aquest document que s'haurà de lliurar al CV en la data establerta.

Exercicis

a) Seccions Paral·leles

Per al codi de l'**apartat-a.c**, creeu una regió paral·lela amb quatre threads que englobi els tres bucles que es proposa modificar. Apliqueu una clàusula que permeti que si el nombre d'elements **N** es menor a 100, l'execució del programa serà completament seqüencial. Dins d'aquesta regió s'han d'assignar els dos primers bucles com a tasques per a threads independents (*sections*). El tercer bucle s'haurà de paral·lelitzar amb un únic pragma amb el treball a repartir entre tots els threads.

i) Mostreu el vostre codi paral·lel. Expliqueu com funciona i que fan cadascuna de les clàusules que heu utilitzat.

1. Especifica que s'executarà una secció en paral·lel de 4 fils sempre que el valor de **N** sigui igual o superior a **100**. Si no, s'executarà en seqüència.
2. Dins d'aquesta secció paral·lela, hi ha dues **seccions** diferents (una per a cada secció) que s'executaran simultàniament en diferents fils.
    a. La primera secció executa un bucle per calcular els valors de **l'array d** basat en l'array b.
    b. La segona secció executa un bucle aniuat per calcular els valors de **l'array c.**
3. Després d'acabar les seccions, s'executa un **bucle for paral·lel** (amb **reducció**) que suma els valors dels arrays d i c en la variable sum_d. Cada fil fa una part de la suma, i utilitza la reducció per combinar els resultats de tots els fils.

En resum, el codi divideix la feina en diverses tasques que es poden executar en paral·lel per millorar el rendiment, sempre que es compleixin les condicions de nombre de fils especificats i el valor de N.

```
#pragma omp parallel num_threads(4) if (N >= 100) Fragment paral·lelitzat (si es compleixen les condicions). Fixat a 4 threads. El if determina si es farà en seqüencial o en paralel segons el valor de N
{

      #pragma omp sections private(i) Execució en paralel per seccions (una per thread). Privatitzem la i ja que comparteixen aquesta variable pel bucle for
   {

      #pragma omp section Executada en un thread qualsevol (no utilitzat per cap altre secció)
      {
         printf("Section 1 is executed by thread %d.\n", omp_get_thread_num());
         for( i = 0; i < N; i++){
            d[i] = i * b[i];
         }
         printf("Finished Section 1 by core %d.\n", omp_get_thread_num());
      }

      #pragma omp section Executada en un thread qualsevol (no utilitzat per cap altre secció)
      {
         printf("Section 2 is executed by thread %d.\n", omp_get_thread_num());
         for( i = 0; i < N; i++){
            for( j = 0; j < N; j++){
               c[i] += a[i] + b[i];
            }
         }
         printf("Finished Section 2 by core %d.\n", omp_get_thread_num());
      }
   } Fi de les  execucions en seccions. Esperen a acabar totes per seguir amb el codi

    #pragma omp for reduction(+:sum_d) For paralel amb reducció de suma (els 4 threads que l'executen comparteixen un resultat en la variable sum_d que al final sumarán fent servir reducció
   for( i = 0; i < N; i++ ){
         sum_d += d[i]+c[i]; Arrays d i c ja acabats de computar en les seccions
   }
   printf("Finished Section 3 by core %d.\n", omp_get_thread_num());
}
```

ii) Les clàusules utilitzades duen alguna espera implícita ? Que implica això ?

Clausales tals com omp section executen les seccions en paral·lel encara que espera que acabin totes per continuar l'execució del codi.

iii) Es possible modificar el codi afegint alguna clàusula d'OpenMP per tal de que els threads no esperin a la finalització dels dos primers bucles per continuar el seu treball ?

Amb la clausula **nowait** es pot seguir executant codi encara que hi hagi un fil executant una secció anterior.

iv) Seria correcte el codi si apliquem la modificació esmentada a l'apartat (iii) ? Per què ?

No, perquè al tercer bucle es necessiten arrays modificats als anteriors bucles i, per tant, hi ha una dependència. Així doncs, no es pot aplicar el **nowait**.

b) **Paral·lelitzant Collatz.**

Considerem la següent funció sobre un nombre natural arbitrari n:

$$f(n) = \begin{cases} n/2 & si\ n\%2 = 0 \\ 3 \times n + 1 & si\ n\%2 = 1 \end{cases}$$

Ara, definim la seqüència següent per qualsevol nombre natural arbitrari n:

$$a_i = \begin{cases} n & per\ i = 0 \\ f(a_{i-1}) & per\ i > 0 \end{cases}$$

La conjectura de Collatz diu que aquesta seqüència acabarà en el valor 1 en un nombre finit de passos (anomenat temps d'aturada total) independentment del nombre natural escollit inicialment.
El següent codi (`collatz.c`) trobar el nombre Y amb el temps d'aturada més alt en el rang [1:X]:

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

unsigned long f(unsigned long x){
   return (x%2) ? 3*x + 1 : x/2;
}

int stop( unsigned long x ){
   int cont = 0;
   while ( x > 1 ){ x = f(x); cont++; }
   return cont;
}

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

   unsigned long n, i, Num;
   unsigned len, Max = 0, id;

   if ( argc < 2 ){ printf("Error: cal indicar el número natural\n" ); exit(1); }

   if ((n = atoi(argv[1])) <= 0 ) { printf("Error: el número ha de ser un natural (> 0)\n" ); exit(1); }

   for ( i = 1; i <= n; i++ ){
      len = stop(i);
      if ( len > Max ) { Max = len; Num = i; };
   }
   printf("El el nombre menor que %lu amb temps d'auturada més alt és %lu amb %u passos\n", n,
Num, Max );
}
```

i) Sembla clar que aquest codi es pot paral·lelitzar amb OpenMP. Com ho faríeu per tal de que aquest codi s'executés en paral·lel fent servir 6 threads, sempre que el **n** sigui més gran o igual a 10^6? (La resposta no és trivial. Adjunteu el codi generat en la vostra resposta).

Per a la paral·lelització d'aquest codi, inicialment declarem la secció paral·lela amb 6 fils, privatitzant la variable "len", ja que aquesta es farà servir per diversos fils en la pròxima secció en paral·lel (bucle for).
A més, incloem la clause shared(Max, Num), ja que aquestes variables estàn relacionades:

- Dins el bucle for en paral·lel, cada fil trobarà una sèrie de longituds (passos) donada la funció stop() segons un número. Si aquest és major al màxim, guardem el valor de la variable "len" a "private_max" i aquest número a "private_num". Així, al final de l'execució tindrem, per cada fil, una variable per la longitud màxima i una del nombre associat a aquesta.
- A la última secció paral·lela (#pragma omp critical), comparem el màxim de cada fil amb el màxim actual (execució en un sol fil). El que fem aquí és obtenir el màxim dels valors màxims de cada un dels 6 fils.
- La raó per la que no fem servir una simple reducció per a quedar-nos amb el valor màxim de passos és que necessitem també el número el qual ha resultat en aquest nombre de passos, i amb la reducció, aquesta informació es perd.

```
int main( int argc, char *argv[] ){
	unsigned long n, i, Num;
	unsigned len, Max = 0, id;

	#pragma omp parallel num_threads(6) private(len) shared(Max, Num) if (n >= 1000000){
		unsigned long private_i;
		unsigned private_max = 0;
		unsigned private_num = 0;

		#pragma omp for
		for (private_i = 1; private_i <= n; private_i++){
			len = stop(private_i);
			if (len > private_max) {
				private_max = len;
				private_num = private_i;
			}
		}
		#pragma omp critical {
		if (private_max > Max) {
			Max = private_max;
			Num = private_num;
		}
		}
	}
	printf("\nEl nombre menor que %lu amb temps d'auturada més alt és %lu amb %u passos\n",
n, Num, Max );
}
```

Adrià Muro Gómez (1665191)
David Morillo Massagué (1666540)

ii) Compileu la versions paral·lela i seqüencial. Feu servir l'eina perf per obtenir el temps d'execució d'ambdues versions pel cas n=20^6 (*perf stat ./<executable> 20000000 [l'script job.sub ja està preparat per aquesta execució]*). Quina acceleració obteniu? Quants processadors diu l'eina perf que heu utilitzat?

Seqüencial : 7,895576032 seconds time elapsed / 1 CPU utilitzades
Paral·lela : 1,536574749 seconds time elapsed / 5,757 CPUs utilitzades

Acceleració = 7,895576032/1,536574749 = 5,1384x

iii) Carregueu ara l'entorn per poder fer servir les utilitats **likwid** (*module load likwid/5.2.2*) i executeu *likwid-perfctr -C 0-5 -g FLOPS_SP ./<executable-omp> 20000000 [l'script job.sub ja està preparat per aquesta execució]*. De tota la sortida que es produeix, observeu la primera taula METRIC (la que dona els resultats per core). En particular l'entrada **Runtime unhalted**. Estan els diferents threads trigant el mateix temps en fer la seva feina? Quin és el percentatge de variació? Que ens pot estar indicant aquesta diferència i per què?

Metric	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5
Runtime unhalted [s]	4.0297	4.4084	4.5493	4.6479	4.6403	4.7667

Veiem que no tots els threads triguen el mateix en acabar, els primers fils triguen menys que els últims. El percertatge de variació que hem obtingut és del 18,29%. El que ens indica això és que el treball està mal repartit. (explitcat en el próxim apartat).

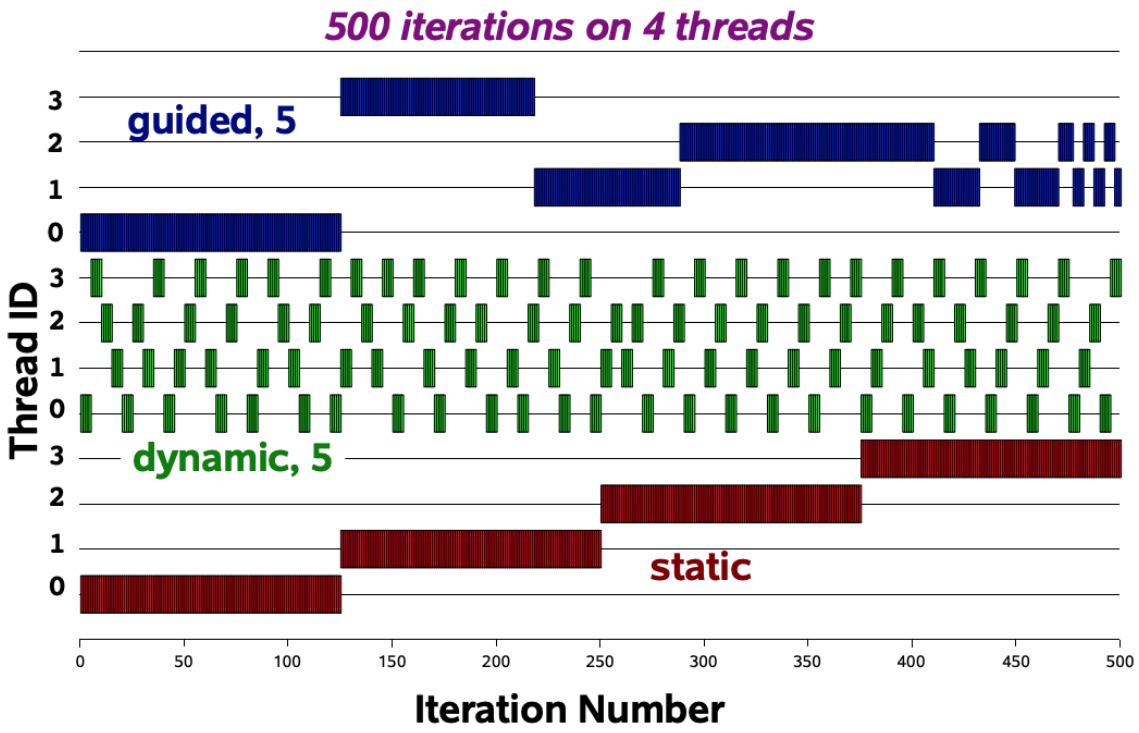
iv) Quina clàusula podem fer servir per millorar encara més el rendiment de la nostra aplicació? (apliqueu-la i indiqueu la millora de rendiment que obteniu)

Fent servir la clàusula **schedule(guided)** en el següent fragment de codi en paral·lel

```
#pragma omp for schedule(guided)
for (private_i = 1; private_i <= n; private_i++){
    len = stop(private_i);
  if (len > private_max) {
    private_max = len;
    private_num = private_i;
}
```

Adrià Muro Gómez (1665191)
David Morillo Massagué (1666540)

D'aquesta manera, es reparteixen les iteracions com veiem en la següent imatge (en color blau):

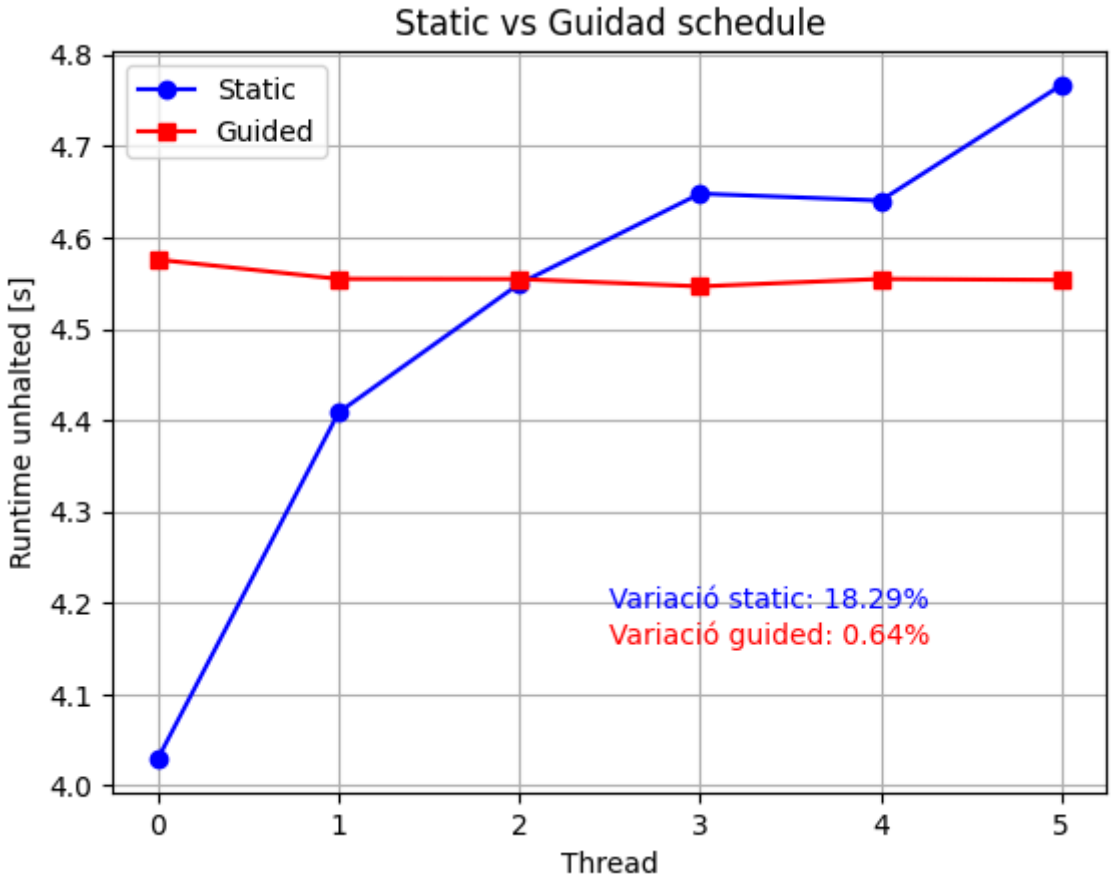


Sense aquesta clàusula, el programa s'executaria fet servir el schedule predeterminat **static**. Aquest últim divideix les iteracions entre el nombre de thread disponibles, i cada un farà 1/n iteracions del bucle. La part negativa de fer-ho així es que, en programes com aquest, les primeres iteracions (nombres menors) tindran un temps de còmput menor que les últimes (per la manera en que es calcula collantz).
Amb l'schedule guiat, evitem que els fils que tenen menys treball acabin abans la seva part de codi i estiguin sense "treballar" esperant als cores que encara estàn computant les seves iteracions. Amb aquesta clàusula, aconseguim millorar l'equilibri de càrrega entre els fils d'execució.
Fent servir el likwid veiem que el treball de cada fil s'ha equilibrat amb els altres:

Metric	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5
Runtime unhalted [s]	4.5756	4.5546	4.5544	4.5464	4.5545	4.5534

Adrià Muro Gómez (1665191)
David Morillo Massagué (1666540)

Hem fet un gràfic fent fervir matplotlib per a visualitzr millor el treball de cada nucli per cada un dels schedules entre Static (blau) i el Guided (vermell).:



Veiem que la variació ha passat d'un 18,29% a un 0,64%, que ens indica que el treball està molt millor respartit que abans.

Executem també les comandes perf stat per compara els temps d'execució:

- Static (default): 4,768749477 seconds time elapsed
- Guided: 4,598076791 seconds time elapsed

A part de repartir millor la feina, aconseguim acabar la feina 0,1 segons abans.