**1 sesion lab2**

//Para copiar la carpeta del lab2 en tu directorio hacer por cada lab

/scratch/nas/1/par0/sessions/lab2.tar.gz .

tar -zxvf lab2.tar.gz

cd pi

geany/nedit pi-v0.c

more Makefile

//Dos versiones con el mismo código una la normal y la otra con chivatos

make pi-v0-omp

make pi-v0-debug //Nos da información

Version Description of changes Correct?

v0 Sequential code. Makes use of omp get wtime to measure execution time yes

v1 Added parallel construct and omp get thread num() no

v2 Added private for variables x and i no

v3 Manual distribution of iterations using omp get num threads() no

v4 Added critical construct to protect sum yes

v5 Added atomic construct to protect sum yes

v6 Private variable sumlocal and final accumulation on sum yes

v7 Use of reduction clause on sum yes

v8 Use of barrier construct yes

//primeras versiones están mal

**v1**: Igual que en el secuencial, no mejora. Pocas iteraciones sacan mal el resultado de pi.

-El resultado está mal porque los dos threads (2) ejecutan todo por eso sale el resultado doble y no se aprovecha el paralelismo.

-También puede ocurrir ‘Data race’.

-Variables definidas antes del ‘#pragma omp parallel’.

**v2**: Se ponen las variables privadas ‘#pragma omp parallel private(i, x)’. Menos la variable sum que no se puede.

**v3**: Ahorra cada uno de los threads se dividen las iteraciones del bucle lo hacemos cambiando el for.

-> for (i=id; i < num\_steps; i=i+num\_threads)

Aún no está bien por la existencia de esa variable sum (data race).

**v4**: Utilizamos ‘#pragma omp critical’ Se ejecuta el código en un solo threads para que no pueda haber data race.

Este código ya está bien.

No obstante la v4 (paralela) tarda muchísimo más que la v0 (secuencial). Por qué?

La sentencia ‘critical’ es muy costosa. Por tanto es inviable.

**v5**: Cambiamos la sentencia ‘critical’ por la sentencia ‘atomic’.

-> #pragma omp atomic

Lo hace por hardware. También cumple la función que cumplía critical ya que hace que el código sea indivisible por cada thread. Atomic se utiliza solo para situaciones concretas.

Mejora mucho el tiempo de ejecución pero no aun lo suficiente, aun v5<v0.

**v6**: Se utiliza una variable sumlocal para cada bucle y despues del bucle se va sumando a una variable sumglobal. Es lo mismo que simular una variable privada.

-> for (i=id; i < num\_steps; i=i+num\_threads) {

x = (i+0.5)\*step;

sumlocal += 4.0/(1.0+x\*x);

}

#pragma omp atomic

sum += sumlocal;

Ahora ya sí que v6>v0.

**v7**: Alternativa de la versión anterior pero ahora de forma implícita con la sentencia

-> #pragma omp parallel private(i, x) reduction(+:sum)

**v8**: Con la sentencia siguiente nos aseguramos que todas las ejecuciones del bucle han sido ejecutadas.

-> #pragma omp barrier

Ahora ir al directorio ‘openmp’ y para cada uno de los 6 archivos contestar las preguntas.

**TODO SOLUCIONADO EN GITHUB**

-> <https://github.com/andyfratello/PAR/tree/main/Laboratori>

**2 Sesion LAB2**

Version Description of changes Correct?

v9 Use of task construct no

v10 Use of single construct to have a single task generator no

v11 Use of taskgroup and reductions (task reduction and in reduction) yes

v12 Use of taskwait and atomic yes

v13 Use of task with dependences (depend clause) yes

v14 Use of taskloop to generate tasks from loop iterations yes

**v9**: Nueva directiva “#pragma omp task private(i,x)” Crea una tasca (explicita) y se ejecuta cuando un procesador está libre. Hasta que no está libre no empieza.

El código es incorrecto porque se duplica la tasca ya que se está ejecutando en paralelo (replicación de tascas).

El sum esta mal (data race).

**v10**: Se añade “#pragma omp single” debajo de la directiva “#pragma omp parallel” para evitar la replicación de tascas que se daba en la anterior versión. Sigue estando mal el sum.

**v11**: Para evitar el data race se añade la directiva

“#pragma omp taskgroup task\_reduction(+: sum)”

(creación de la tasca se marca que se quiere hacer reduccion), debajo de “#pragma omp single” y abajo se añade

“#pragma omp task private(i,x) in\_reduction(+: sum)” (ejecucion de la tarea se marca que quieres hacer reducción también)

El código es correcto.

**v12**: Es otra opción de la v11 (mejor la 11). Se añade encima de la suma de pi la directiva “ #pragma omp taskwait” (Hacer la otra).

**v13**: En vez de hacer la reducción utilizamos la directiva

“#pragma omp task private(i,x) depend(out: sum1)” para cada una de las dos tascas (se guardan en variables locales por lo que no puede haber data race) y al final se hace la suma

#pragma omp task depend(in: sum1, sum2) (Espera a que acabe las tascas “sum1” y ”sum2”)

sum += sum1 + sum2;

Esta version es igual de correcta que la v11.

**v14**: Se utiliza la directiva

“#pragma omp taskloop private(x) reduction(+: sum) num\_tasks(8)” // grainsize(num\_steps/8)”

Antes siempre “#pragma omp parallel” y “#pragma omp single”

En esta versión solo tiene un bucle por lo que nos ahorramos directivas. (Fácil y eficiente).

num\_tasks(número de tascas deseadas) crea las tascas que le introduzcas y se divide el trabajo.

Si solo pones #pragma omp parallel son tascas implícitas en cambio si tu pones explícitamente el taskloop entonces son tascas explicitas.