Problemes PSD: HPC.

Conceptes bàsics HPC

1)

1.1)

a) $T(1)$ és la suma de tots els temps de les tasques del graf.

Per tant, $T(1) = 7 \cdot 5 + 15 = 50$ u.

$T(\infty)$ és el temps amb infinits recursos, i és $T(\infty) = 3 \cdot 5 + 15 = 30$ u.

El camí crític és T1, T5, T11, T12.

b) El paral·lelisme és $\frac{T(1)}{T(\infty)} = \frac{50}{30} = \frac{5}{3}$.

1.2)

a) Consisteix en repartir la càrrega de treball que hi ha en total entre tots els nodes del graf de tasques: això per exemple es podria traduir en repartir la feina que s'ha de fer en paral·lel (~~per mirinyacs~~) entre tots els threads que tenim per a minimitzar el temps que trigarem a completar el camí crític.

b) Donada una càrrega total de treball, si tenim molts taques molt petites, el camí crític tindrà un cost menor per dur-les a terme, però podríem tenir un cost de creació (overhead) molt gran. En canvi, si tenim tasques més grans, però en tenim menys, el camí crític tindrà un cost no òptim, però no tindrem problemes d'overhead.

2)

a) $T(1) = 9 \cdot 5 + 2 \cdot 10 + 15 = 80$

b) El camí crític és T1, T5, T9, T11, T12. En formen part aquestes tasques i no d'altres, perquè amb les dependències que generen i el temps que triguen juntes superen a qualsevol altra combinació.

c) $T(\infty) = 4 \cdot 5 + 15 = 35$.

d) $P_{\text{paral·lelisme}} = \frac{T(1)}{T(\infty)} = \frac{80}{35} = \frac{16}{7}$.

3)

a) La fracció paral·lela del programa és tot el que cau dins la regió marcada per #pragma omp parallel excepte per les línies perdudes per #pragma omp single, que són el init_A i init_B. Per tant, podem paral·lelitzar els dos bucles, és a dir, un total del 99.7% del programa. Per tant, $\boxed{\phi = 0.997}$.

b) El màxim speedup que podríem obtenir seria, segons la llei d'Amdahl, $\frac{1}{1-\phi} = \frac{1}{0.003} = \frac{1000}{3}$.

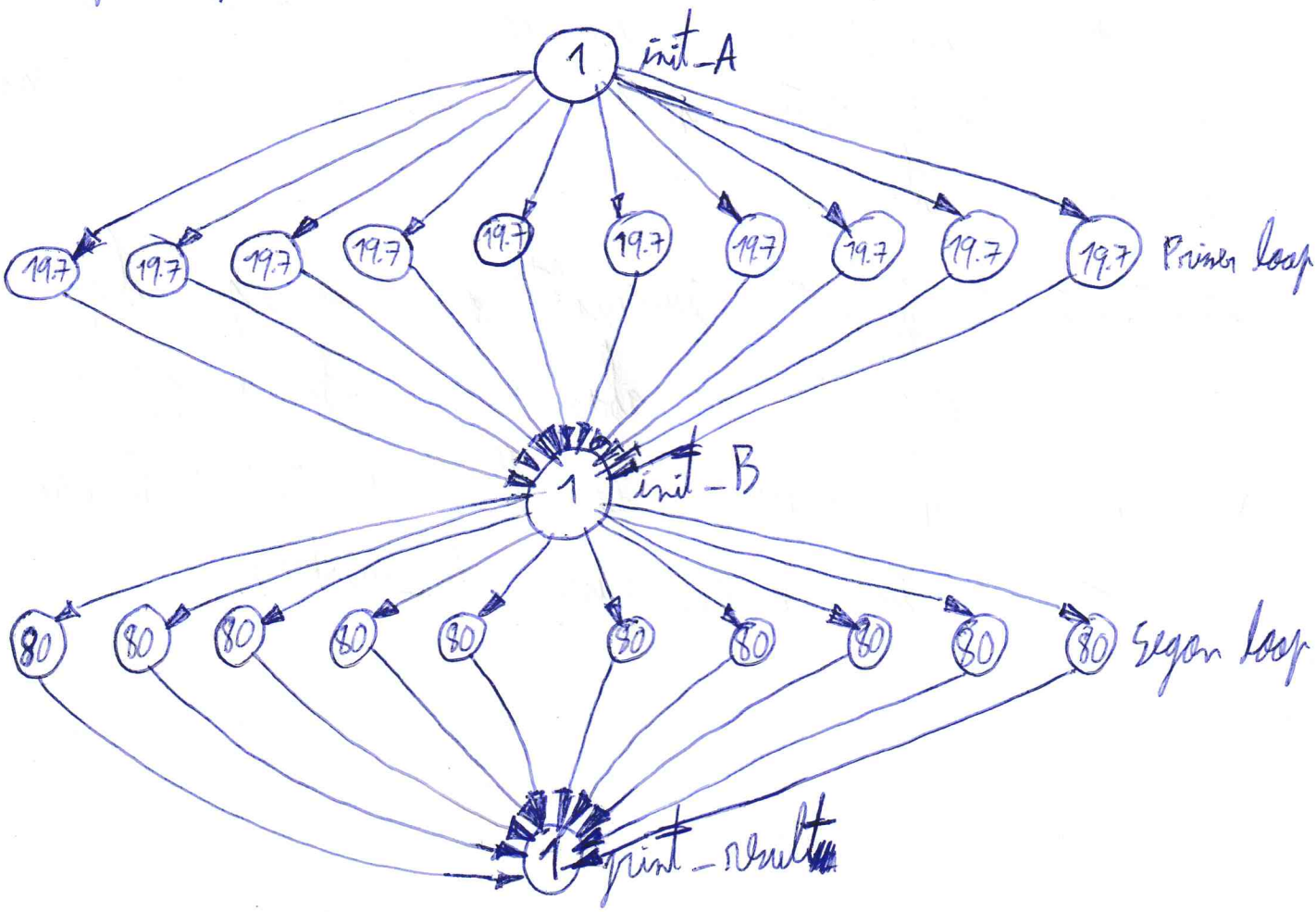
c) Tant en el primer bucle com en el segon hauríem de privatitzar la variable j, però no la i. Això és degut a que la i ja la fa sola, però la j no. En no estar privatitzada, aquesta última podria generar conflicte entre threads.

d) En executar el codi amb 48 ~~threads~~ cores ens trobarem amb que no estarem aprofitant al màxim la ~~capacitat~~ capacitat que tenim, és a dir, l'eficiència de la paral·lelització serà molt baixa. (Estarem threads idle). Per solucionar això, podríem col·locar els dos bucles a les zones repetitives (càlculs lents)

pragma omp for collapse (2).

Així, estarem repartint DIM_X · DIM_Y = 20 · 1000000 = 2 · 10^7 iteracions entre 48 cores i serà més fàcil que l'eficiència augmenti amb el nombre de cores en general.

e) Graf de tasques amb 10 threads, escala perfectament:



f) El temps $T(10)$ és $T(10) = 1 + 19.7 + 1 + 80 + 1 = 102.7$ segons.

Per tant, $Speedup(10) = \frac{T(1)}{T(10)} = \frac{1000}{102.7} \simeq 9.737$.

g) $Eficiència(10) = \frac{Speedup(10)}{10} \simeq 0.9737$.

OpenMP

1) Hem vist tres clàusules principalment:

- shared: és una clàusula redundant, marca una variable com a compartida entre els threads. Aquesta forma de visibilitat és la que hi ha per defecte per qualsevol variable declarada abans de la regió paral·lela.
- Private: crea una còpia de la variable marcada per cada thread. Cada thread modifica només la seva còpia de la variable, i el valor inicial d'aquesta és aleatori i no depèn del valor de la variable abans de la regió paral·lela.
- Firstprivate: té la mateixa funcionalitat que private, però el seu valor inicial és el mateix que abans d'entrar a la regió paral·lela.

2) Com que hi ha un collapse(2), cada thread executarà 10 iteracions i per tant, 10 instàncies de la funció do_work().

3)

a) El codi calcula la suma dels elements d'una matriu A quan se li aplica una funció f a cadascun. Es creen dos threads, i cadascun d'ells crearà 10 tasques. Pel thread amb id=0, les tasks creades faran iteracions:

- iter0_49: farà 50 iteracions, de la 0 a la 49 (inclosa).
- iter50-99: " " , de la 50 a la 99 (inclosa).
- iter100_149: " " , 100-149.
- iter150_199: " " , 150-199.
- iter200_249: " " , 200-249.
- iter250_299: " " , 250-299.
- iter300_349: " " , 300-349.
- iter350_399: " " , 350-399.
- iter400_449: " " , 400-449.
- iter450_499: " " , 450-499.

El thread amb id=1 crearà les tasques:

- iter500_549
- iter550_599
- iter600_649
- iter650_699
- iter700_749
- iter750_799
- iter800_849
- iter850_899
- iter900_949
- iter950_999

b) Al final del parallel (línia 32) hi ha una sincronització implícita, i a la línia 28 hi ha una sincronització explícita (atomic).

c) No podem saber quin thread executarà cada task, ja que no sabem els costs de creació/destrucció, i l'assignació de tasks a cada thread es fa dinàmicament, és a dir, que un thread agafa una task quan està lliure.

d) Sí, ja que és una variable compartida. A més el seu valor serà correcte ja que està protegida per una clàusula atomic.

MPI

1) Una operació col·lectiva en MPI és una comunicació entre N processos a la vegada a través d'un comunicador.

2) En un gather, un procés rebrà informació de cada procés del seu comunicador. En canvi, en un Allgather, tots els processos rebran informació de tots els altres en el mateix comunicador, i per tant, tots enviaran també alguna informació.

3) Podem esperar 4 línies amb la frase "var_compartida = ?", on ? val dir que per cada procés aquesta variable tindrà per valor el número del thread que hi hagi arribat més tard.

4) El resultat

a) [illegible] no serà correcte perquè les comunicacions que es fan serà asíncrones i, per tant, podria ser que un dels processos mirats tingués les dades.

b) No solucionaria el problema, ja que MPI_Wait només s'assegura que el primer procés enviï les dades, però no s'assegura que es rebin. Tot podrien solucionar fent comunicació síncrona.

c) No solucionaria el problema, tampoc, ja que el barrier només s'assegura que els processos arriben allí i s'hi esperen abans de continuar, no de que les crides anteriors hagueixin sigut exitoses.

d)

```
void compute_dimensions(int my_size, int my_rank, int size,
int * init, int *end, int *chunk)
{
    chunk = size / my_size;
    init = my_rank * chunk;
    end = init + chunk;
}
```

5)

a) És un grup de processos MPI. El comunicadors s'utilitzen per transmetre informació entre processos. Un mateix procés pot estar en tants comunicadors com es vulgui.

b) Una comunicació síncrona és aquella que s'espera a que el procés que ha de rebre la informació l'hagin rebut per continuar amb el programa.

c) Una comunicació entre N processos a la vegada a través d'un comunicador.

d) Un broadcast envia tota la informació als processos objectius, mentre que un scatter separa la informació i n'envia un tros a cada procés objectiu.

e) Com que a MPI no hi ha variable compartides, cada procés imprimirà el seu propi PID. Per tant, tindrem 48 PIDs diferents.