PAR PARCIAL

Parallel computing

SERIAL VS PARALLEL (videolesson 1)

- Serial: T = N/F
- Parallel: dividir el programa en parts discretes (tasques) i usen CPU per executar-les en el mateix temps

PARALLEL
Shared memory: suport hw per una bona compartció de memòria i sincronització.
Distributed-memory: suport hw per l'accés remot de dades i comunicació.
Idealment cada procés: 1/P del programa -> T = (N/P) /F
Throughput: multiples instruccions no relacionades s'executen al mateix temps en múltiples processadors. Si k programes en P processos, cada programa rep P/k processors

CONCURRENCY AND PARALLELISM

Execució concurrent: trencar el problema en parts (tasques), cada tasca s'executa sequencialment en una sola CPU però múltiples tasques intercalen la seva execució en la CPU
Execució parallela: usar múltiples CPU per executar tasques simultàneament, 1 programa p CPU, cada CPU té 1/p del programa

PROBLEMES

- **Data Race Condition:** múltiples tasques llegeixen i escriuen alguna variable i el resultat final depèn del moment relatiu de la seva execució.
- **Deadlock:** dos o més tasques no poden seguir amb l'execució perquè una està esperant que alguna de les altres facin algo
- **Starvation:** una tasca no pot guanyar accés a un recurs compartit i no pot avançar.
- **Livelock:** dos o més tasques estan canviant constantment el seu estat de resposta als canvis en altres tasques sense fer res útil.

PROCESSORS, THREADS AND PROCESSES

Processes/threads: agents computacionals LÒGICS, oferts pel SO, que executen tasques
Processors/CPU: unitats de HARDWARE que executen fisicament els threads.
No necessàriament ha d'haver una relació 1-1 entre Threads i CPU

Parallelism

TASKS AND THEIR IMPLICATIONS

Task decomposition: en funció al procediment a fer (funcions, loops...)
Data decomposition: en funció a les dades a tractar (vector, matrius, columnes...)
VIDEO LESSON 2
TDG: directed acyclic graph. Representa les dependències
T1 = suma dels costos de totes les tasques
Tinf = suma dels costos de les tasques que formen el camí critic
Parallelism = T1/Tinf
Pmin = mínim nombre de processadors necessàris per aconseguir Parallelism
Tp = T1/P però amb overhead -> Tp = T1/P +ovh(P)

GRANULARITY AND OVERHEADS
Granularity: la mida de les tasques.

El paral·lelisme creix quan hi ha tasques més petites.
Fine-grained tasks: ex: només un element o dos del vector, cada iteració d'un loop
Coarse-grained tasks: ex: tot un vector, tot el loop

Per saber si hi ha dependències entre tasques hem de veure com s'executaría en seqüencial. Si llegeixo valors que han estat modificats anteriorment -> dep

T = Tcomp + Tsync (ovh) = (#taks col+ #task fila-1)*Ttask + (#taks col + #task fila-2)*Tsync

SPEED-UP AND EFFICIENCY

VIDEO LESSON 3
Tp = temps d'execució en p CPU
Speed-up: Sp = T1/Tp
Speed-up parallel part = T part parallela normal / T part paralela paral·lelitzada

- Tp: T sequencial + T part paral·lela paral·lelitzada

Eficiència: Ep = Sp/P (Speed-up amb P proc / #proc)
Ep = P -> 100% eficient. ef > 1 -> usant P processadors el programa pot executar-se P vegades més ràpid,
ef < 1 -> el típic speedup es inferior al linear.
ϕ: PARALLEL FRACTION
T1 = Tseq+Tpar,
ϕ = Tpar/T1,
T1 = (1-ϕ)*T1 + ϕ*T1
Tp = Tseq+Tpar/P, Tp = (1-ϕ)*T1 + (ϕ*T1/P) (+overhead(p))
Sp = 1/((1-ϕ)+ϕ/P)
Sinf = 1/1-ϕ
Tp ≥ T1/P ideal case
Tp ≥ Tinf crytical path

Embarrassingly parallel: **no existeixen dependències** entre les tasques originades a la task decomposition. Totalment parallel.
Load unbalance: una de les fonts de degradació del rendiment en l'execució paralela.

Strong scaling: incrementar CPU **mantenint la mida** del problema. **Reduir el temps** d'exec.
Weak scaling: incrementar CPU amb un problema de **mida proporcional** a P. Solventar **problemes més grans.**

SOURCES OF OVERHEAD:

- DATA SHARING: **explicit** a través de missatges o implícit a través de caches
- IDLENESS: un thread no pot trobar **cap feina útil** que executar
- COMPUTATION: s'afegeix **feina extra** per **obtenir** l'algorisme de **paral·lelisme**
- MEMORY: s'usa **memoria extra** per **obtenir** l'algorisme de **paral·lelisme**
- CONTENTION: **competició** per l'accés dels **recursos compartits.**

Tacces = ts + m*tw, Tx = Tcomput + Tacces // **Tcomput = Tinf si Pcomput = Pmin**

- Ts: temps d'start-up
- m: numero de bytes transferits. Dades que agafo d'un altre processador.
- Tw: temps per byte transferit

CONDICIONS en un moment donat:

- Pi només pot executar un accés remot a memòria
- Pi només pot servir un accés remot a memòria
- Pi només pot executar un accés a Pj i servir un altre de Pk

Tp = (n/c + P-1)*(n/P *c)*tbody + (ts+n+tw)+((n/c)+P-2)*(ts+c*tw))

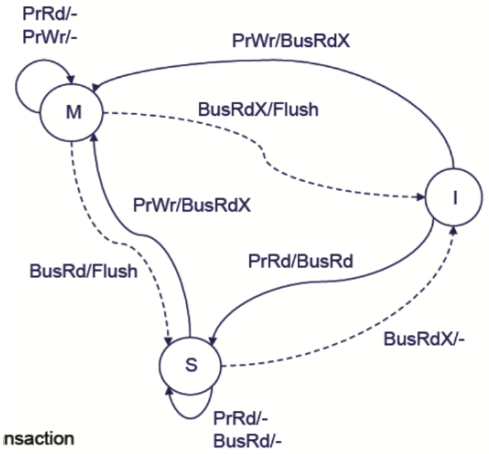
Parallel architectures

UMA architectures (video lesson 4)

UMA: Uniform Memory Access (Time)
SMP: Symmetric MultiProcessor: tots els cores amb les seves caches es veuen igual en tot el sistema
Write Update coherence: per cada **escriptura** s'ha d'enviar a totes les **altres caches** que poden tenir el **valor i actualizar-lo**
Write Invalidate coherence: després d'una **escriptura** a una de les cache, totes les **cache** que tenien una **copia** del valor s'han **d'invalitzar** i actuaran com si fos un miss el següent cop que vulguin accedir a la dada
Snooping: totes les **escriptures són broadcast** al bus compartit i l'ordre en el que apareixen en el bus és l'ordre en el que s'han de veure per tothom
Directory: diferent ordenació de punts es poden usar en diferents blocs.

Bits per mantenir la coherència al sistema INSIDE numa node:
(#processadors *) [#linies * 2bits/linia, #linies = mida MC/ mida linia caché]
MSI: Modified (dirty copy, only one), Shared (clean copy), Invalid
MESI, E: exclusive (only one clean)
CPU events: PrRd, PrWr
Bus events: BusRd, BusRdX, BusUpgr, Flush
Fetch: busco components externs
True sharing: l'intercanvi de dades és inevitable en parallel computing. Hi ha mecanismes de coherència per permetre aquesta compartició de dades; la sincronització permet compartir adequadament.
False sharing:

- Hi ha línies de la caché que també introdueixen artefactes: més d'un data object diferent o múltiples elements del mateix, que resideixen la mateixa línia de caché
- FS passa quan diferents processadors fan referència (R and W) als diferents objectes de la mateixa línia de caché.



NUMA architectures (video lesson 5)

NUMA: Non-Uniform Memory Access
El **temps** d'accés és variable, **depèn dels #salts que hagi de fer**
Snooping: un sol bus on van totes les peticions dels cores. El bus es converteix en coll d'ampolla. No funciona bé amb més de 8-16 cores
Snoopy-invalidate: Quan escrius algo, invalides els següents perquè no es puguin llegir ni escriure.
Non-broadcast network: sobre la qual es van enviant les peticions.

- Distributed across cores: no centralitzat en un bus. No totes les peticions han d'anar a la mateixa part del Directory.

Bits per mantenir la coherència AMONG numa node:
(#nodes)* [(Mida MP/ Mida linia de MP) * (2+Presence bits/linia)]

Home node: node en el qual es troba la linia. "First Touch".
Local node: node amb el processador accedint a la línia
Remote node: owner tenint copies dirty o lector tenint copies clean de la linia.
Commands local -> home: RdReq, WrReq, UPgrReq
After RdReq/WrReq Home -> local: Dreply (clean copy of line)
After UpgrReq Home -> local: Ack (to give permission)
Commands home -> remote: Fetch (torna Dreply), invalidate (torna ack)

Transpas

SIMD: Single Instruction Multiple Data: DLP (data-level parallelism)

Memory hierarchy:

- LOCALITAT:
    - Temporal: la dada es tornarà a accedir aviat (loop)
    - Espaial: accedir a les adreces del costat (vector, matriu)
- LÍNIA/BLOC: Paraules consecutives en memòria (32B = 4 words x 8B)
- ACCÉS:
    - Hit: la dada esa a una de les línies d'aquell nivell.
    - Miss: la dada s'ha de recuperar d'una linia del següent nivell.
- ALGORISMES DE REEMPLAÇAMENT: lru, fifo, lfu….
- WRITE POLICY: write-through, write-back (on hit), write-allocate, write-no-allocate (on miss)

ITERATIVE TASK DECOMPOSITION

- Usage of taskloop construct with a granularity bigger than 1, for instance 2 or evenly distributing the iterations among threads (num_tasks(omp_get_num_threads()))
- Usage of reduction clause to avoid synchronizations to update the max and min durations
- Add constructs parallel and single in the main program

RECURSIVE TASK DECOMPOSITION

- Implement a recursive tree task decomposition
- Add a new parameter to count the depth level
- Usage of final and mergebale clauses to implmement the cut-off based on the recursivity tree level.
    - **#pragma omp task final (condicio) mergeable**
    - Condició =
- Force min_max1 and min_max2 to be shared and a taskwait to wait for the two created tasks to be finished
- Add constructs parallel and single in the main program.
- BS = NUM_ELEMENTS_PER_CACHE_LINE/2

INPUT BLOCK GEOMETRIC DATA DECOMPOSITION:

Requereix sincronització en l'update del output del vector del qual s'estigui fent la DD. Per reduir aquesta sync a 1 (només al final), s'ha d'usar una variable temporal tmp on anar acomulant els resultats dins dels inner loops: **scalar replacement**.

- Declarar com a private les variables que calgui // **#pragma omp parallel private(var)**
- Variables necessaries:
    - id = omp_get_thread_num();
    - nt = omp_get_num_threads();
    - bs = C/nt;
    - reminder = C%nt;
    - extra = id < reminder, extraprev = extra? id : reminder;
    - lowerb = id*bs + extraprev
    - upperb = lowerb + bs + extra
- for d'accés normal, i = 0, i < R ++i // j= loweb, j < upperb, ++j i abans d'actualitzar vector **#pragma omp atomic**

OUTPUT BLOCK-CYCLIC GEOMETRIC DATA DECOMPOSITION:

- Definir número d'elements per linia de cache
- #pramgma omp parallel private(variables)
- Variables que necessitem: id, nt, BS = num_elements_per_cache_line, loweb = id*BS, step = nt*BS
- For per fer salts de blocs (ii = loweb, ii < R; ii+=step)
    - For per navegar per cada bloc (i = ii, i < min(R, ii+BS), i++)
        - codi del for normal amb scalar replacement

FUNCIO MIN

#define min(a,b) ((a) > (b) ? (b) : (a))

IMPLICIT TASK

```
void func(...){
      #pragma omp parallel{
            int thid = omp_get_thread_num();
            int nt = omp_get_num_threads();
            int bs = n/nt;
            for(int i = thid*bs; i < (thid+1)*bs; i++)...
}     }
```

EXPLICIT TASKS WITH TASK

```
void func(...){
      for(int i = 0; i < n; i++){
            #pragma omp task
}     }
void main(){
      #pragma omp parallel
      #pragma omp single
      func();
}
```

EXPLICIT TASKS WITH TASK AND GRANULARITY CONTROL

```
void func(...){
      int thid = omp_get_thread_num();
      int nt = omp_get_num_threads();
      int bs = n/nt;
      for(int ii = 0; i < n; i+=bs){
            #pragma omp task
            for(int i = ii; i < min(ii+BS, n), i++) ..
}     }
```

EXPLICIT TASKS WITH TASKLOOP

```
void func(...){
      int nt = omp_get_num_threads();
      int bs = n/nt;
      #pragma omp taskloop grainsize(bs)
      for(int i = 0; i < n; i++) ...
}     }
```

ATOMIC

```
void func(...){
      #pragma omp taskloop
      for(int i = 0; i < n; i++){
            #pragma omp atomic
            result += ...
}     }
```

REDUCTION

```
void func(...){
      #pragma omp taskloop reduction(+: variable)
      for(int i = 0; i < n; i++) variable = …;
}
```

CRITICAL (només primer normal, només segon optimitzat)

```
void func(...){
      #pragma omp taskloop
      for(int i = 0; i < n; i++){
```

```
            #pragma omp critical
            if(...)
            {
                  #pragma omp critical
                  ...
}     }     }
```

NAMED CRITICAL

```
void func(...){
      #pragma omp taskloop
      for(int i = 0; i < n; i++){
            if(x % 2){
                  #pragma omp critical(parell)
                  compute(result.even)
            }
            else {
                  #pragma omp critical(senar)
                  compute(result.odd)
}     }     }
```

LOCKS

```
void func(...){
      #pragma omp taskloop
      for(int i = 0; i < n; i++){
            índex = ...
            omp_set_lock(&x[index]);
            ...
            omp_unset_lock(&x[index]);
}     }
void main(){
      for(i = 0; i < ...; i++) omp_init_lock(&x[i]);
      #pragma omp parallel
      #pragma omp single
      func(...)
      for(i = 0; i < n; i++) omp_destroy_lock(&x[i]);
}
```

ITERATIVE TASK DECOMPOSITION

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(BS)
for(i = 0; i < num_elem; i++){
      //com sigui
}
```

INPUT BLOCK DATA DECOMPOSITION

```
#pragma omp parallel private(variables) {
      int thid = omp_get_thread_num(); int nt = omp_get_num_threads();
      int BS = num_elem/nt; int lowerb = thid*bs; int upperb = lowerb + bs;
      if(thid == nt-1) upper = num_elem;

      for(i = lowerb; i < min(upperb, num_elem); i++){
            //codi de dins del for de iterative task decomposition.
}     }
```

INPUT CYCLIC DATA DECOMPOSITION

```
#pragma omp parallel private(variables) {
      int thid = omp_get_thread_num(); int nt = omp_get_num_threads();
```

```
      for(i = thid; i < num_elem; i+=nt){
            //codi de dins del for de iterative task decomposition.
}     }
```

INPUT BLOCK-CYCLIC DATA DECOMPOSITION

//bs numero d'elements per bloc

```
#pragma omp parallel private(variables) {
      int thid = omp_get_thread_num(); int nt = omp_get_num_threads();

      for(int ii = thid*bs; ii < num_elem; ii+=nt*bs){
            for(i = ii; i < min(ii+BS, num_elem); i++){
                  //codi de dins del for de iterative task decomposition.
}     }     }
```

OUTPUT BLOCK DATA DECOMPOSITION

```
#pragma omp parallel private(variables) {
      int thid = omp_get_thread_num(); int nt = omp_get_num_threads();
      int BS = num_elem/nt; int lowerb = thid*bs; int upperb = lowerb + bs;
      if(thid == nt-1) upper = num_elem;

      for(i = 0; i < num_elem; i++){
            //constant
            if( (x ≥ lowerb) and (x < upperb)) //algo més
}     }
```

OUTPUT CYCLIC DATA DECOMPOSITION

```
#pragma omp parallel private(variables) {
      int thid = omp_get_thread_num(); int nt = omp_get_num_threads();

      for(i = 0; i < num_elem; i++){
            //constant
            if( (x % nt) == thid) //algo més
}     }
```

OUTPUT BLOCK-CYCLIC DATA DECOMPOSITION

//bs número d'elements per bloc

```
#pragma omp parallel private(variables) {
      int thid = omp_get_thread_num(); int nt = omp_get_num_threads();

      for(i = 0; i < num_elem; i++){
            //constant
            if( ((x/bs) % nt) == thid) //algo més
}     }
```

RECURSIVE DIVIDE AND CONQUER WITHOUT CUT-OFF

```
... rec_func(int elem,...){
      if(elem == 1) { ...}
      else{
            n = elem/2;
            #pragma omp task shared(var1)
            var1 = rec_func(n, ...);
            #pragma omp task shared(var2)
            var1 = rec_func(elem-n, ...);
            #pragma omp taskwait
      }
```

```
      return (var1+var2);
}
```

RECURSIVE DIVIDE AND CONQUER WITH CUT-OFF

```
... rec_func(int elem,..., int level){
      if(elem == 1) { ...}
      else if (!omp_in_final()){
            n = elem/2;
            #pragma omp task shared(var1) final(level ≥ MAX_LEVEL) // final(elem < vs)
            var1 = rec_func(n, ... , level+1);
            #pragma omp task shared(var2) final(level ≥ MAX_LEVEL)
            var1 = rec_func(elem-n, ... , level+1);
            #pragma omp taskwait
      }
      else{
            var1 = rec_func(n, ... , level+1)
            var2 = rec_func(elem-n, ..., level+1)
      }
      return (var1+var2);
}
```