Guió i material per classe no presencial PAR grup 20
Dilluns 3/5/2021, 10:00-12:00

Problema 1 del Tema 4

SAXPY (*Single-precision A times X Plus Y*) is a combination of scalar multiplication and vector addition. It takes as input two vectors of 32-bit floats x and y with n elements each, and a scalar value a. It multiplies each element $x[i]$ by a and adds the result to $y[i]$, as shown below:

```
void saxpy(int n, float a, float *x, float *y) {
        for (int i = 0; i < n; ++i) y[i] = a * x[i] + y[i];
}
```

(a) *version 1*: with implicit tasks executed by threads created in the `parallel` region.

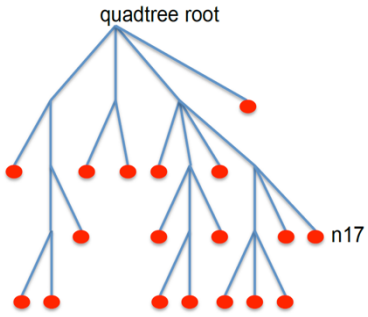
```
void saxpy(int n, float a, float *x, float *y) {
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    int  P = omp_get_num_threads();
    int BS = n/P;
    for (int i = id*BS; i < (id+1)*BS; ++i) y[i] = a * x[i] + y[i];
  }
}
```

(b) *version 2*: with explicit tasks but without `taskloop`.

```
void saxpy(int n, float a, float *x, float *y) {
  #pragma omp parallel
  #pragma omp single
  {
    int P = omp_get_num_threads();
    int BS = n/P;
    for (int ii = 0; ii < n; ii+=BS)
      #pragma omp task
      for (int i = ii; i < ii+BS; ++i) y[i] = a * x[i] + y[i];
  }
}
```

(c) *version 3*: with explicit tasks with `taskloop`.

```
void saxpy(int n, float a, float *x, float *y) {
  #pragma omp parallel
  #pragma omp single
  {
    int P = omp_get_num_threads();
    int BS = n/P;
    #pragma omp taskloop grainsize(BS)
    for (int i = 0; i < n; ++i) y[i] = a * x[i] + y[i];
  }
}
```



Problema 3 del Tema 4

Tree: L'enunciat permet diverses respostes vàlides, en funció de l'ordre de creació de les tasques en cada nivell de recursivitat. Si aquestes es creen d'esquerra a dreta, llavors la resposta és (3 + 4 + 3) · t = 10 · t. Si és de dreta a esquerra, llavors la resposta correcta és (2 + 1 + 1) · t = 4 · t.

Leaf: L'enunciat permet diverses respostes vàlides, en funció de l'ordre d'invocació de les crides en cada nivell de recursivitat. Si aquestes es criden d'esquerra a dreta, llavors la resposta és 17 · t (nombre de fulles a l'esquerra de n17, ella inclosa). Si és de dreta a esquerra, llavors la resposta correcta és 2 · t (nombre de fulles a la dreta de n17, ella inclosa).

Condicions de carrera en task decompositions (I): data sharing constraints

Fem un parell de canvis en el codi original de la setmana passada per tal que el càlcul en la funció `doComputation` 1) faci una acumulació sobre una variable global, o 2) retorni un resultat. Comencem amb el primer cas, tal com es mostra a continuació per la implementació *recursive leaf task decomposition*:

```
int result = 0;

void doComputation (int * vector, int n) {
   for (int i = 0; i < n; i++)
      result += compute (&vector[i]);
}

void partition (int * vector, int n) {
   if (n > MIN) {
      int size = n / 4;
      for (int i = 0; i < 4; i++)
         partition (&vector[i*size], size);
   } else
      #pragma omp task                          // vector and n firstprivate
      doComputation (vector, n);
}

void main () {
   #pragma omp parallel
   #pragma omp single
   partition (vector, N);
}
```

És correcte el codi? No, fixeu-vos que les tasques que s'executen en paral·lel poden provocar una condició de carrera en l'accés a la variable compartida `result` sobre la que s'acumula el resultat de la funció `compute`. Però això ja sabeu com protegir-ho: amb atòmic, o amb més overhead, amb `critical`.

```
void doComputation (int * vector, int n) {
   for (int i = 0; i < n; i++)
      #pragma omp atomic
      result += compute (&vector[i]);
}
```

Amb atomic només un thread tindrà accés a la variable compartida result en un instant de temps. Si ho fem sobre la variable real, tindrà molt overhead, doncs seràn n atomics per tasca.

L'atomic protegeix l'accés a una variable (lectura /escriptura). Té una estructura rígida: variable += 1 -> llegeixo, opero i escric. Per això és més eficient

I això sabem que pot portar massa overhead, que podem minimitzar fent privatització de variables (slide 49):

```
void doComputation (int * vector, int n) {
   int tmp = 0;
   for (int i = 0; i < n; i++)
      tmp += compute (&vector[i]);

   #pragma omp atomic
   result += tmp;
}
```

Si la variable global no es necessita fins al final de la regió paralela, podem crear una variable privada (declarada dins al cos de la tasca) i utilitzar-la per les tasques com podem veure al codi de l'esquerra.

Llavors hem de fer l'atomic per cada acomulació final sobre la variable global, així reduïm a 1 atomic per tasca en comptes de n atomics per tasca.

Slide 47

En aquesta slide tenim el resum de les tres variants principals del `#pragma omp atomic`:

- `update`: lectura/escriptura atòmiques;
- `read`: només lectura atòmica;
- `write`: només escriptura atòmica;

Si no es fica cap de les tres clàusules, s'aplica per defecte `update`. Per exemple:

```
void doComputation (int * vector, int n) {
    for (int i = 0; i < n; i++)
        if (vector[i] == VALUE)
            #pragma omp atomic write
            result = i;
}
```

Recordeu que l'`atomic` no sempre el podem fer servir, i llavors tenim que anar a `critical`:

```
void doComputation (int * vector, int n) {
    for (int i = 0; i < n; i++)
        #pragma omp critical
        if (vector[i] < result)
            result = vector[i];
}
```

El critical protegeix tot el codi que encapsula. Protegeix una regió de codi. Assegura que la regió l'executarà només 1 sol thread. L'utilitzem quan no podem fer servir l'atomic per culpa de no seguir l'estructra rígida que té. És menys òptim.

Els processadors es queden esperant abans del critical. Intenten entrar i només entren a la regió crítica quan no hi ha cap processador treballant-hi.

ja que necessitem que la regió `critical` que inclogui la comparació que es fa en el condicional i l'actualització en cas de complir-se la condició (estem buscant el valor mínim del vector). Podríem en aquest cas reduir l'overhead del `critical` de manera que no es fes en cada iteració?

```
void doComputation (int * vector, int n) {
    for (int i = 0; i < n; i++)
        if (vector[i] < result)
            #pragma omp critical
            if (vector[i] < result)
                result = vector[i];
}
```

Per a optimitzar-ho podem dobler la condició. Si la condició no es cumpleix passo de llarg i així no he d'esperar al critical. Si no ho fem així un processador es quedarà esperant al critical encara que quan entri no faci res sobre la variable perquè la condició no es cumpleix.

de manera que només fem el `critical` en cas de saber que hi ha opcions d'actualitzar el valor mínim trobat fins el moment.

Slide 48

I en aquesta veiem la possibilitat d'afegir-hi un nom a la regió `critical`. Per exemple aquí tindríem un possible cas d'ús:

Posar nom al critical ens serveix per a poder tenir varies regions protegides independents entre elles.

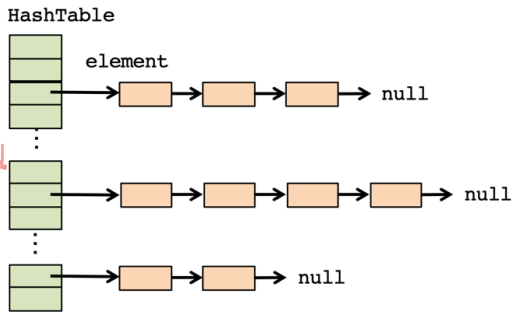
```
void doComputation (int * vector, int n) {
    for (int i = 0; i < n; i++)
        if (vector[i]%2)          // l'element del vector es parell
            #pragma omp critical(parell)
            compute(result.even);
        else                      // l'element del vector es senar
            #pragma omp critical(senar)
            compute(result.odd);
}
```

Abans de passar a l'estratègia tree, anem a veure un exemple més en el que farem la inserció de l'element del vector en una taula de hash (slide 52):

```
#define SIZE_TABLE 1048576
typedef struct {
    int data;
    element * next;
} element;
element * HashTable[SIZE_TABLE];

void doComputation (int * vector, int n) {
    for (i = 0; i < n; i++) {
        int index = hash_function(vector[i]);
        insert_element (vector[i], index);
    }}
```

determina l'index dins la taula de hash



#pragma omp critical

Llista ordenada sequencial que s'ha de protegir. Al ser ordenada no podem fer-ho com volguem.

En aquest cas l'execució de la funció `insert_element` s'ha de fer en exclusió mútua, però no el càlcul de la funció `hash_function` que determina on hem d'inserir. Com que és una crida a una funció ens veiem forçats a fer servir `critical`:

```
void doComputation (int * vector, int n) {
    for (i = 0; i < n; i++) {
        int index = hash_function(vector[i]);
        #pragma omp critical
        insert_element (vector[i], index);
    }
}
```

Ara només 1 thread pot inserir en TOTA la taula. Però insert_element treballa a nivell d'entrada (index)! Per tant el que és realment sequencial és la incersió en una de les llistes de la hashtable.

Per tant podem fer múltiples incersions en paralel sempre i quan siguin en diferents entrades de la hashtable. En aquest cas estem sobreprotegint!

Observeu que estem limitant molt el paral·lelisme del problema, ja que només hauríem de protegir en cas de voler inserir sobre la mateixa entrada (llista) de la `hash_table`. Com ho podem fer? Amb `critical(name)` no serveix doncs ens hauríem d'inventar `SIZE_TABLE` noms diferents i fer ús d'un condicional per diferenciar-los a tots.

Slide 51

En aquesta slide mostrem un tipus de variables especials que ofereix OpenMP, els anomenats locks (tipus de variable `omp_lock_t`) , que poden pendre dos possibles valors: ocupat o lliure. I dos possibles operacions sobre ells: agafar (`omp_set_lock`) i alliberar (`omp_unset_lock`). A l'iintentar-lo agafar, si el lock està ocupat, s'espera que s'alliberi; si està lliure, l'agafa i el marca com ocupat. A l'executar la funció d'alliberar, simplement el marca com a lliure.

Hi ha una tercera operació que simplement pregunta si el lock està lliure o no (`omp_test_lock`). I els locks s'han d'inicialitzar abans de fer-ne ús (`omp_init_lock`) i destruir (`omp_destroy_lock`) quan ja no es facin servir.

Amb aquests tipus de variable podem escriure una versió molt més eficient que l'anterior, permeten que es puguin actualitzar múltiples entrades de la taula de hash en paral·lel (slide 53):

```
void doComputation (int * vector, int n) {
    for (i = 0; i < n; i++) {
        int index = hash_function(vector[i]);
        omp_set_lock(&hash_lock[index]);
        insert_element (vector[i], index);
        omp_unset_lock(&hash_lock[index]);
    }
}
```

Declarem un vector hash_lock que acompanya la taula de hash. Per a cada entrada existeix un lock asssociat que protegirà que només un thread actualitzi l'entrada index de la taula de hash.

Un cop tenim index intentem posar un lock (si ja està lock esperarem fins que estigui lliure). Un cop acabem traiem el lock.

i abans en algun moment s'hauran creat i s'hauran de destruir, per exemple en el `main`:

```
void main () {
   for (i = 0; i < HASH_TABLE_SIZE; i++) omp_init_lock(&hash lock[i]);

   #pragma omp parallel
   #pragma omp single
   partition (vector, N);                    // N is multiple of P

   for (i = 0; i < HASH_TABLE_SIZE; i++) omp_destroy_lock(&hash lock[i]);
}
```

Condicions de carrera en task decompositions (II): task ordering constraints

Què hauria passat si la funció `doComputation` no actualitzes directament la variable global, sinó que retornes com a resultat acumulat de les crides a la funció `compute`? Anem a veure-ho en la *recursive tree task decomposition*, que hem re-escrit una mica per fer-ho més clar:

```
int result = 0;
int doComputation (int * vector, int n) {
   int tmp = 0;
   for (int i = 0; i < n; i++)
      tmp += compute (&vector[i]);
   return(tmp);
}

int partition (int * vector, int n) {
   int tmp1=0, tmp2=0, tmp3=0, tmp4=0;

   if (n > MIN) {
      int size = n / 4;                          // n is multiple of 4
      #pragma omp task                           // vector and size firstprivate
      tmp1 = partition (&vector[0], size);
      #pragma omp task
      tmp2 = partition (&vector[size], size);
      #pragma omp task
      tmp3 = partition (&vector[2*size], size);
      #pragma omp task
      tmp4 = partition (&vector[3*size], size);
   } else
      tmp1 += doComputation (vector, n);

   return(tmp1+tmp2+tmp3+tmp4);
}

void main () {
   #pragma omp parallel
   #pragma omp single
   result = partition (vector, N);
}
```

* Cada crida recursiva defineix una nova tasca.

Ara no actualitzem result. El resultat el retorna la funció.

Partition també retorna el resultat de sumar les 4 tmps. En cas de ser una fulla serà tmp1+0+0+0.

No puc sumar i sortir fins que no tingui el valor de tmp1, tmp2, tmp3 i tmp4.

també podem utilitzar un taskgroup.

Com veieu, així ens estalviaríem fer un atòmic en cada iteració del bucle, treballant sobre una variable local. En aquest cas la crida en el cas base hauria de recollir el resultat i fer que la funció recursiva `partition` també el retornes. Però, hem d'afegir algun tipus de sincronització? Si, entre les tasques filles que calculen un resultat i la tasca pare que les està creant i necessita el seu resultat. Com?

El shared l'hem de posar perquè cada una de les variables tmpx son les mateixes que les variables tmpx generals.

Si no poso el shared, el omp task em farà un firstprivate i no ho volem ja que volem que la tasca pare comparteixi la variable amb les filles.

```
int partition (int * vector, int n) {
   int tmp1=0, tmp2=0, tmp3=0, tmp4=0;

   if (n > MIN) {
      int size = n / 4;                          // n is multiple of 4
      #pragma omp task shared(tmp1)              // vector and size firstprivate
      tmp1 = partition (&vector[0], size);
      #pragma omp task shared(tmp2)
      tmp2 = partition (&vector[size], size);
      #pragma omp task shared(tmp3)
      tmp3 = partition (&vector[2*size], size);
      #pragma omp task shared(tmp4)
      tmp4 = partition (&vector[3*size], size);
      #pragma omp taskwait
   } else
      tmp1 += doComputation (vector, n);

   return(tmp1+tmp2+tmp3+tmp4);
}
```

Taskwait vs Taskgroup:

El taskwait es una barrera on la tasca pare s'espera a que totes les tasques filles acabin. Totes les tasques que ha creat. Això NO inclou les filles de les filles (netes). Només espero a les tasques meves.

El taskgroup en canvi espera a TOTES les tasques. Les filles, les netes, vesnetes, etc. Totes les child i els seus descendents. El taskgroup a més comença en un punt i acaba en un altre ("{}"). Marquem la regió de la qual volem esperar les tasques.

Dues coses: 1) el `taskwait` assegura que tenim el resultat de totes les tasques filles abans de retornar la seva suma com a resultat de la crida recursiva; això no em fa perdre per res el paral·lelisme que tenim ja que es una *tree decompostion*. I 2), per què hem de ficar la clàusula `shared` per les variables `tmp1`, `tmp2`, `tmp3` i `tmp4`? Doncs per que per defecte aquestes variables es farien `firstprivate` i no em retornarien el resultat que necessitem.

Slides 54 i 55

Recordeu del Lab2 que `taskwait` no és l'única manera que tenim per expressar *task barriers*, també tenim el `taskgroup`. Veiem les diferències en l'exemple d'aquesta slide. Mireu els dos exemples; bàsicament, `#pragma omp taskwait` espera a què totes les tasques filles fins aquell punt acabin, però no les descendents de les tasques filles. En canvi `#pragma omp taskgroup` defineix una regió de codi en la que s'espera al final a què totes les tasques descendents (filles, nétes, ...) acabin.

Slides 56 i 57

Donat un TDG, com el podem expressar en OpenMP. En les dues slides es mostren diferents opcions d'expressar-lo amb taskwait/taskgroup, per cadascuna d'elles veient quines limitacions en el paral·lelisme obtingut ens introdueixen vs. la complexitat d'escriure la sincronització adient. L'ultima opció que dona la slide 56 fa ús de task dependences, molt més elegant i obtenint el paral·lelisme esperat en el TDG. Anem a veure com expressar-les.

Slide 58

Les *task dependences* ens permetran definir relacions d'ordre entre tasques: clàusula `depend`. Amb l'ús d'aquesta clàusula podem especificar el que necessita (`in`) i el que genera (`out`) l'execució d'una `task`, o també la `inout` combinada. El que s'especifica són variables.

Slide 59

I que impliquen aquestes clàusules? Doncs una `in` sobre una variable `var` fa que la tasca s'esperi a qualsevol tasca creada prèviament que tingui la mateixa variable `var` com a `out` (individual o combinada en forma de `inout`): Això té una mica més overhead! Només s'utilitza quan hi ha dependències entre tasques.

```
// variables a i b son shared
#pragma omp task depend(out:a)
a = foo1();
#pragma omp task depend(in:a) depend(out:b)
b = foo2(a);
#pragma omp task depend(in:b)
foo3(b);
```

aquesta tasca genera una sortida anomenada "a"

aquesta tasca necessita l'entrada "a" i genera l'entrada "b"

Per ultim aquesta necessita "b"

És la típica dependència productor→consumidor. I una `out` sobre una variable `var` que provoca? Doncs a què qualsevol tasca prèvia que tingui la mateixa variable `var` com a `in` o com a `out` acabi. Perquè?

```
#pragma omp task depend(out:a)
a = foo1();
#pragma omp task depend(in:a)
foo2(a);
#pragma omp task depend(out:a)
a = foo3();
```

Observeu que si `foo3` s'executés abans que la `foo2`, aquesta `foo2` podria no agafar el resultat que ha generat la `foo1`, que és el que realment necessita. És una dependència deguda al re-ús de la variable `a`. I aquest altre cas:

```
#pragma omp task depend(out:a,b)
        {
        a = foo1();
        b = foo2(a);
        }
#pragma omp task depend(out:a)
a = foo3();
#pragma omp task depend(in:b)
foo4(b);
#pragma omp task depend(in:a,b)
foo5(a);
```

Sense dubte `foo5` s'espera a que la tasca que executa `foo3` acabi per la `a`, ja que s'esperarà a l'última tasca que genera la variable `a`, i a la `foo1/foo2` per la `b`, Però que passaria si abans que `foo5` comenci l'execució, pel motiu que sigui, també acaba l'execució de `foo1`? Doncs que llavors `foo5` agafaria el valor generat per `foo1` i no el de `foo3`. Per evitar això, `foo3` no pot començar fins que `foo1/foo2` acabin; evidentment, això és degut al fet de voler reaprofitar la variable `a`.

Slides 61 a 63 (OPCIONALS)

Aquestes slides mostren funcionalitats addicionals que us comentem breument per veure que existeixen, però no per fer-les servir en els problemes:

- Possibilitat que `taskwait` s'esperi només a unes determinades tasques, amb la clàusula `depend`.
- Com expressar múltiples dependències en una única clàusula: multidependències. Insert text here
- Fer menys restrictives les cadenes de dependències que es poden originar amb `inout`: `mutexinoutset`.

Problema 6 del Tema 4

a) Primer de tot, observeu que no hi ha cap regió paral·lela `#pragma omp parallel`. Per tant, si no hi ha threads, no hi ha cap mena de paral·lelisme per mes `#pragma omp task` que es facin servir. I segon, quin tipus de recursive task decomposition s'està fent servir? Leaf, ja que es genera una nova tasca en arribar a cada fulla de l'arbre de recursivitat; això vol dir que totes les tasques es creen de forma seqüencial per part del thread que entri per (o en altres paraules, la tasca implícita que executi) el `#pragma omp single`.

b) Implementem per tant una tree recursive task decomposition sense preocupar-nos en aquest apartat del control de cut-off que ens demanen després.

```
void quicksort_base(int * v, int n); // sorts a vector v of n elements
int find_pivot(int *v, int n);       // finds the index n of the pivot

void quicksort(int *v, int n) {
   int index;
   if (n<N_BASE)
        quicksort_base(v, n);
   else {
        index = find_pivot(v, n);
        #pragma omp task
        quicksort(v, index);
        #pragma omp task
        quicksort(&v[index], n-index);
}}

void main() {
   #pragma omp parallel
   #pragma omp single
   quicksort(v, N);
}
```

c1) Afegim ara el mecanisme de cut-off basat en nivell de recursivitat, que, si es vol implementar de forma eficient, implica afegir un argument addicional a la funció, tal com es va veure l'altre dia a les slides:

```
#define MAX_DEPTH 5
void quicksort(int *v, int n, int level) {
   int index;

   if (n < N_BASE) quicksort_base(v, n);
   else {
      index = find_pivot(v, n);
      if (level < MAX_DEPTH) {
         #pragma omp task
         quicksort(v, index, level+1);
         #pragma omp task
         quicksort(&v[index], n-index, level+1);
      } else {

         quicksort(v, index, level);              // could also be level+1
         quicksort(&v[index], n-index, level);    // could also be level+1
      }
   }
}

void main() {
   ...
   #pragma omp parallel
   #pragma omp single
   quicksort(v, N, 0);
   ...
}
```

Segurament haureu pensat com fer-ho sense l'argument addicional, però és costos en temps d'execució, ja que requereix fer un $\log_2$. En concret podríem saber el nivell en què ens trobem simplement fent nivell = log2(N/n).

Versions alternatives fent ús de `final` i `omp_in_final`:

```
index = find_pivot(v, n);
if (!omp_in_final()) {
   #pragma omp task final(level >= MAX_DEPTH)
   quicksort(v, index, level+1);
   #pragma omp task final(level >= MAX_DEPTH)
   quicksort(&v[index], n-index, level+1);
} else {
   quicksort(v, index, level);              // could also be level+1
   quicksort(&v[index], n-index, level);    // could also be level+1
}
```

La tasca es marca com a final quan es compleix la condició dins el final. Llavors podem comprovar-ho amb el omp_in_final().

O `final` i `mergeable`:

```
index = find_pivot(v, n);
#pragma omp task final(level >= MAX_DEPTH) mergeable
quicksort(v, index, level+1);
#pragma omp task final(level >= MAX_DEPTH) mergeable
quicksort(&v[index], n-index, level+1);
```

S'especifica la condició i amb el mergeable es fa que no es posi dins el sac de tasques. Evitem posar l'if.

c2) Similarment, el cut-off basat en la mida del vector, deixant de generar tasques quan s'arriba a una mida que es considera massa petita (poca granularitat per la tasca):

```
#define VECTOR_SIZE 128
void quicksort(int *v, int n) {
   int index;
   if (n < N_BASE) quicksort_base(v, n);
   else {
      index = find_pivot(v, n);
      if (!omp_in_final()) {
         #pragma omp task final(n < VECTOR_SIZE)
         quicksort(v, index);
         #pragma omp task final(n < VECTOR_SIZE)
         quicksort(&v[index], n-index);
      } else {
         quicksort(v, index);
         quicksort(&v[index], n-index);
      }
   }
}

void main() {
   ...
   #pragma omp parallel
   #pragma omp single
   quicksort(v, N);
   ...
}
```

Per la propera classe:

1) Entendre els diagrames temporals en les slides 56 i 57 i com es garanteixen les dependències entre tasques
2) **Problemes a fer:** 8 (repàs iterative i recursive amb cut-off); 10 apartats a) i b) (critical vs. locks); problemes 15 i 16 (locks); problemes 11 i 12 (task dependences).