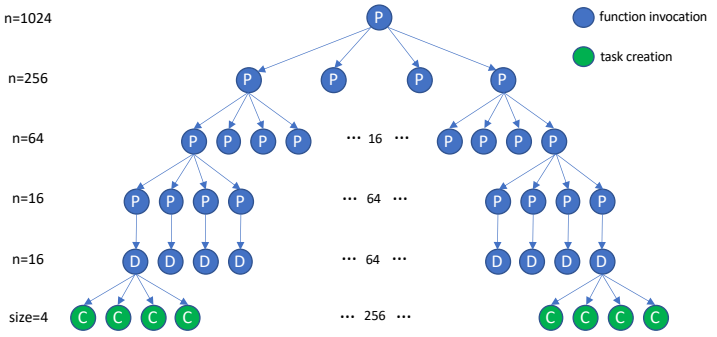
Guió i material per classe no presencial PAR grup 20
Dilluns 26/4/2021, 10:00-12:00

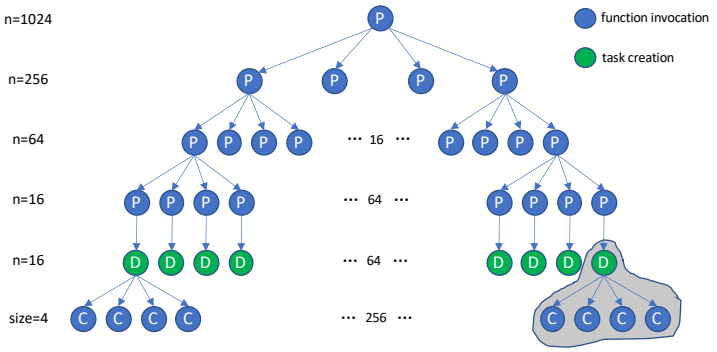
Repàs problema Atenea

```
#define N 1024
#define MIN 16

void doComputation (int * vector, int n) {
   int size = n / 4;
   for (int i = 0; i < n; i += 4)
      tareador_start_task("compute");
      compute(&vector[i], size);
      tareador_end_task("compute");
}

void partition (int * vector, int n) {
   if (n > MIN) { // MIN is multiple of 4
      int size = n / 4;
      for(int i=0; i<4; i++)
         partition(&vector[i*size], size);
      }
   else
      doComputation(vector, n);
   return;
}
void main() {
   ...
   partition (vector, N); // N is multiple of 4
   ...
}
```



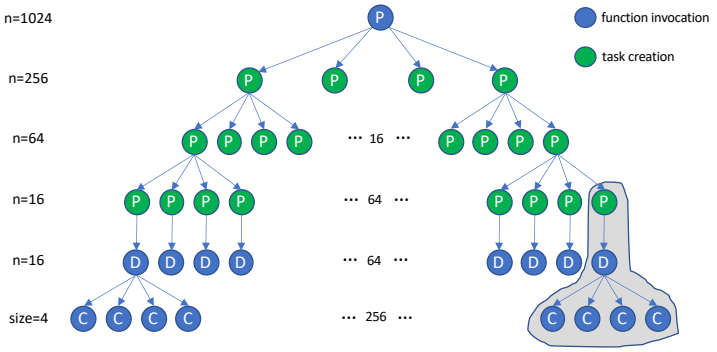
#tasks = 256, granularity = 1, T_∞ = (256 x 2) + 16 = 528

```
#define N 1024
#define MIN 16

void doComputation (int * vector, int n) {
   int size = n / 4;
   for (int i = 0; i < n; i += 4)
      compute(&vector[i], size);
}

void partition (int * vector, int n) {
   if (n > MIN) { // MIN is multiple of 4
      int size = n / 4;
      for(int i=0; i<4; i++)
         partition(&vector[i*size], size);
      }
   else
      tareador_start_task("doComputation");
      doComputation(vector, n);
      tareador_end_task("doComputation");
   return;
}
void main() {
   ...
   partition (vector, N); // N is multiple of 4
   ...
}
```



#tasks = 64, granularity = 4, T_∞ = (64 x 2) + (4 x 16) = 192

```
#define N 1024
#define MIN 16

void doComputation (int * vector, int n) {
   int size = n / 4;
   for (int i = 0; i < n; i += 4)
      compute(&vector[i], size);
}

void partition (int * vector, int n) {
   if (n > MIN) { // MIN is multiple of 4
      int size = n / 4;
      for(int i=0; i<4; i++)
         tareador_start_task("partition");
         partition(&vector[i*size], size);
         tareador_end_task("partition");
      }
   else
      doComputation(vector, n);
   return;
}
void main() {
   ...
   partition (vector, N); // N is multiple of 4
   ...
}
```



#tasks=84, granularity=4, T_∞ = (3 x 4 x 2) + (4 x 16) = 88

Repàs de les iterative task decomposition

Com expressaríem la iterative task decomposition del quizz en OpenMP? Anem a fer el codi una mica més general per tal de poder treballar bé amb les diferents formes d'expressar les descomposicions en tasques. **Suposarem que `N` i `MIN` són múltiples del nombre de processadors.** Tenim diverses opcions, segons volem fer servir tasques implícites o explícites:

```
void doComputation (int * vector, int n) {
   for (int i = 0; i < n; i++)                 // n is multiple of P
      compute (&vector[i]);
}

void partition (int * vector, int n) {
   if (n > MIN) {                              // MIN is multiple of P
      int size = n / 4;
      for (int i = 0; i < 4; i++)
         partition (&vector[i*size], size);
   } else
      doComputation (vector, n);
   return;
}

void main () {
   partition (vector, N);                      // N is multiple of P
}
```

a) iterative task decomposition amb implicit tasks

```
void doComputation (int * vector, int n) {
   #pragma omp parallel
   {
   int whoamI = omp_get_thread_num();
   int howmany = omp_get_num_threads();
   int size = n / howmany;                     // n is multiple of P
   for (int i = whoamI*size; i < (whoamI+1)*size; i++)
      compute (&vector[i]);
   }
}
```

Cada thread executarà una tasca implícita que fa un únic bloc de `size` iteracions. No hi ha cap possibilitat de canviar la granularitat de les tasques si volem mantenir un bon balanceig de càrrega.

b) iterative task decomposition amb explicit tasks (versió `task`)

```
void doComputation (int * vector, int n) {
   for (int i = 0; i < n; i++)                 // n is multiple of P
      #pragma omp task                         // vector and i firstprivate
      compute (&vector[i]);
}


void partition (int * vector, int n) {
   if (n > MIN) {                              // MIN is multiple of P
       ...
   } else
      doComputation (vector, n);
}

void main () {
   #pragma omp parallel
   #pragma omp single
   partition (vector, N);                      // N is multiple of P
}
```

Observeu que cada tasca explícita executarà una única iteració (granularitat 1); podem fer granularitats diferents? Sí, però com abans, hem de reestructurar el bucle a mà:

```
doComputation (int * vector, int n) {
   int BS = ...;                                  // BS is the granularity
   for (int ii = 0; ii < n; ii += BS)
      #pragma omp task                            // vector, ii and BS firstprivate
      for (int i = ii; i < min((ii+BS), n); i++)
         compute (&vector[i]);
   }
}
```

c) iterative task decomposition amb explicit tasks (versió `taskloop`)

El control de la granularitat es pot fer més simple fent ús de la construcció `taskloop`:

```
void doComputation (int * vector, int n) {
   int BS = ...;                                  // BS is the granularity
   #pragma omp taskloop grainsize(BS)             // vector firstprivate by default
   for (int i = 0; i < n; i++)
      compute (&vector[i]);
}
```

o també amb `num_tasks(n/BS)`.

Si amb taskloop és tan senzill, per que parlem de fer-ho amb tasques implícites? Les tasques implícites son estàtiques, en quant a que sabem quin thread les executa i podem fer el repartiment de feina d'acord amb això. Veurem en el Tema 5 que això ens permetrà controlar localitat de les dades, quelcom que ja sabem que és important en arquitectures paral·leles. Per contra, les tasques explicites son dinàmiques i no hi ha cap control sobre qui les executa.

Recursive task decompositions

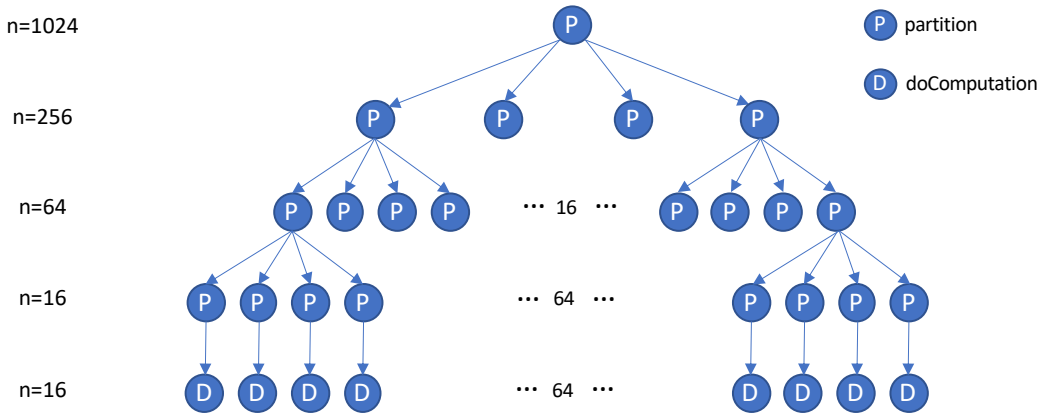
Deixem els bucles enrere i passem a l'altre tipus de task decomposition que esteu treballant en el laboratori actual (Lab4) i que també heu vist en l'últim vídeo d'Atenea: les **recursive task decomposition**. Les crides recursives són una segona font de tasques paral·leles molt important.

Continuarem amb el mateix exemple que estem treballant en aquesta classe; a les slides podeu trobar el mateix, però fet amb un codi que fa el producte intern de dos vectors `A` i `B` amb una estratègia recursiva.

```
void partition (int * vector, int n) {
   if (n > MIN) {                                 // MIN is multiple of P
      int size = n / 4;
      for (int i = 0; i < 4; i++)
         partition (&vector[i*size], size);
   } else
      doComputation (vector, n);
}

void main () {
   partition (vector, N);                         // N is multiple of P
}
```

En cada invocació de la funció `partition` es fan 4 crides recursives a la mateixa funció, fins que arriba al cas base i llavors executa la funció `doComputation` que hem vist abans.

Des de `main` es fà la primera crida a `partition` amb mida `N` (és a dir, el problema senser). Cadascuna de les invocacions de `partition` divideix el problema de mida `n` en 4 subproblemes de mida `n/4`, fent una crida recursiva a `partition` amb la quarta part dels elements, i així fins a arribar a la mida de vector de `MIN`, moment en què crida a la funció `doComputation`. Aquesta estratègia de resoldre el problema s'anomena "**divide and conquer**".



Quantes fulles té l'arbre de recursivitat? N/MIN=1024/16=64. En quants passos de recursivitat es fan? En $\log_4(N/MIN)= \log_4(64)=3$, on la base del logaritme (4) és el nombre de crides recursives per nivell. Si féssim 2 crides per nivell, seria $\log_2(64)=6$ passos de recursivitat.

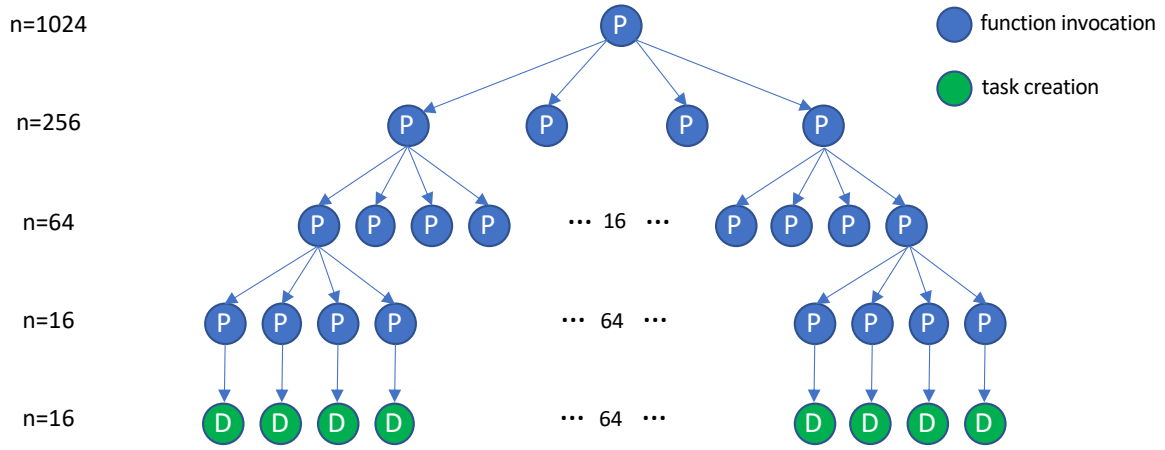
Recursive leaf task decomposition

Com ja sabeu, dues possibilitats de cara a definir les tasques. La primera s'anomena **leaf decomposition**, que tal com diu el seu nom consisteix a generar tasques a l'arribar a les fulles de l'arbre de recursivitat, en el nostre exemple, una tasca per executar la funció `doComputation`, el cas base de la recursivitat.

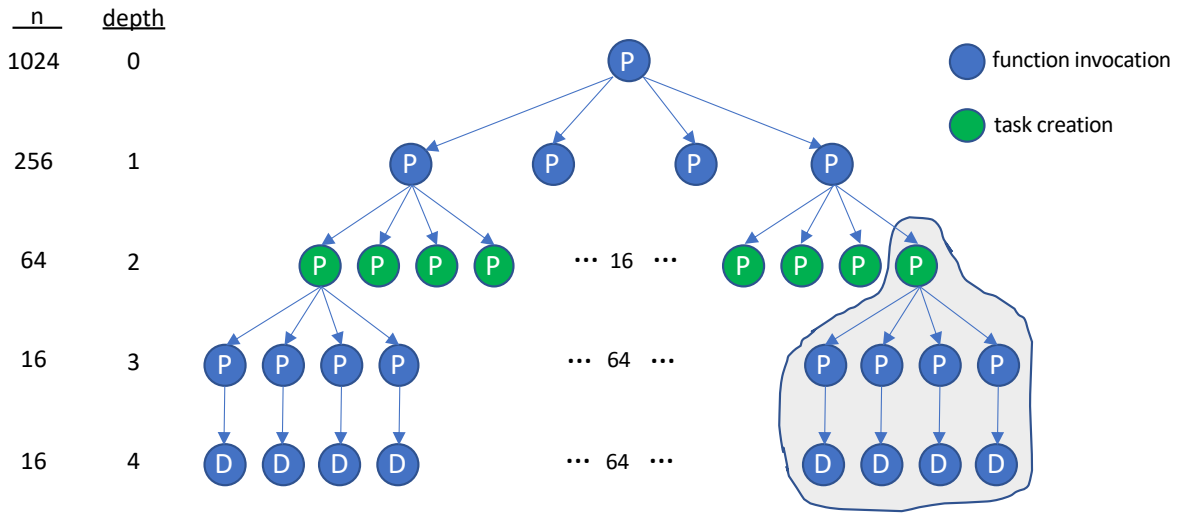
```
void partition (int * vector, int n) {
   if (n > MIN) {                              // MIN is multiple of P
      int size = n / 4;
      for (int i = 0; i < 4; i++)
         partition (&vector[i*size], size);
   } else
      #pragma omp task                         // vector and n firstprivate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

void main () {
   #pragma omp parallel
   #pragma omp single
   partition (vector, N);                      // N is multiple of P
}
```

Quants threads generen tasques en aquest cas? 1, el que entra en el `single`. Quins threads executen tasques? Tots!, primer els P-1 que no entren en el single i quan aquest acabi, també contribuirà en la seva execució. Per tant recordem que el problema aquesta estratègia és que les **tasques es generen de forma seqüencial**. No hi ha paral·lelisme en el procés de generació

de tasques, ja que com veieu, el thread que entra en el single és el responsable de generar totes les 64 tasques.



Quina és la granularitat de les tasques generades? Podríem dir que és 1 fulla de l'arbre de recursivitat per tasca (és a dir l'execució de `doComputation` amb mida `MIN`). Podem canviar aquesta granularitat per fer-la més gran? Sí, si provoquem la generació de tasques a un nivell inferior de la recursivitat, abans d'arribar a les fulles, per exemple:



A aquest nivell de recursivitat l'anomenem **nivell de cut-off**. Cada vegada que el recorregut seqüencial de l'arbre de recursivitat arribi al nivell de cut-off es generaran tasques, cadascuna acabarà de recórrer el seu subarbre de recursivitat que té per sota. En l'exemple, generem 16 tasques (en lloc de 64), cadascuna amb una granularitat 4 vegades més gran (és a dir, 4 invocacions de `doComputation` cadascuna de mida `MIN`).

En el codi és tan senzill com afegir un condicional que determini si estem en el nivell de cut-off o no, i per això necessitem saber en quin nivell de recursivitat ens trobem. Això ho podem fer afegint un argument més a la funció, la variable `depth`, que inicialment val 0 en la crida des de `main` i es va incrementant d'1 en 1 en cada nivell de crides recursives. Així en el condicional simplement haurem de preguntar per si el valor de la variable `depth` és igual a `CUT-OFF`, que en aquest cas està inicialitzada a 2.

```
#define CUT_OFF 2

void partition (int * vector, int n, int depth) {
   if (n > MIN) {                              // MIN is multiple of P
      int size = n / 4;
      if (depth == CUT_OFF)
         for (int i = 0; i < 4; i++)
            #pragma omp task
            partition (&vector[i*size], size, depth+1);
      else
         for (int i = 0; i < 4; i++)
            partition (&vector[i*size], size, depth+1);
   } else
      doComputation (vector, n);
}

void main () {
   #pragma omp parallel
   #pragma omp single
   partition (vector, N, 0);                          // N is multiple of P
}
```

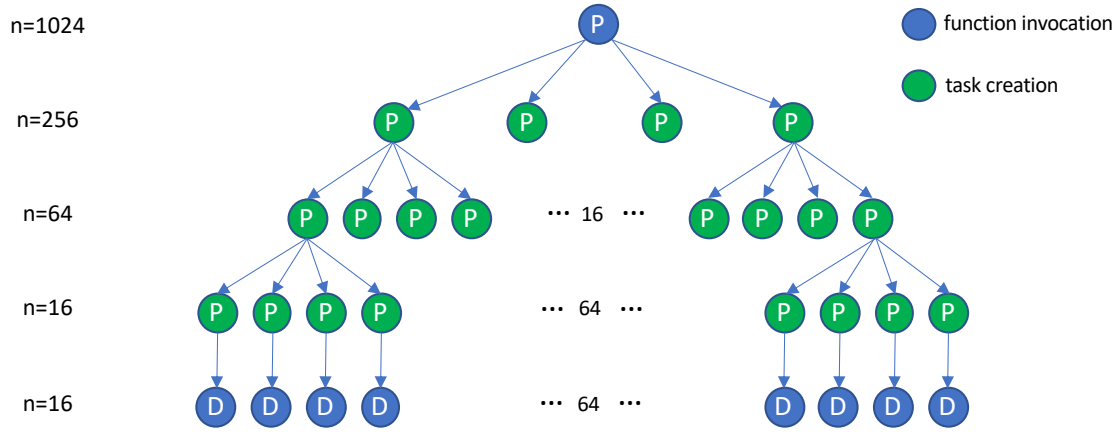
Fixeu-vos que el condicional que controla el cas base de la recursivitat (`n>MIN`) i el condicional que controla el cut-off (`depth==CUT_OFF`) són diferents. El primer decideix si arribem o no al cas base, i en cas negatiu, el segon controla si generem tasques o no.

Recursive tree task decomposition

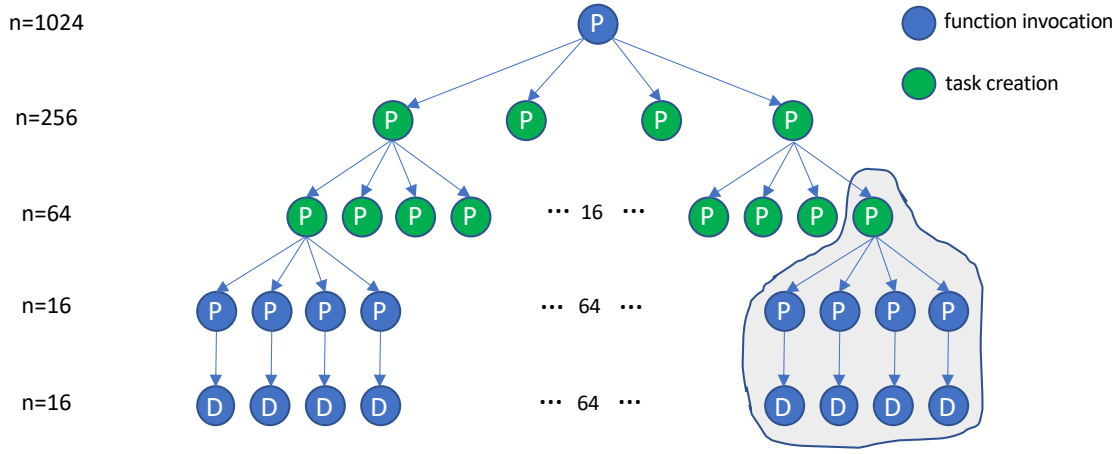
Anem a veure a continuació l'estratègia dual anomenada **recursive tree decomposition**. Amb aquesta estratègia, per cada branca de l'arbre de recursivitat es genera una tasca, és a dir, cada crida recursiva a `partition` es defineix com a tasca.

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

void main () {
   #pragma omp parallel
   #pragma omp single
   partition (vector, N);                      // N is multiple of P
}
```

Quin efecte té això? Si mirem en l'arbre de recursivitat veurem que en la primera invocació des de `main` de la funció `partition` es creen 4 tasques. Aquestes 4 primeres tasques es podran executar en paral·lel, de manera que cadascuna d'elles tornarà a invocar a `partition` que en generarà 4 més, en paral·lel. I així fins a arribar al cas base, moment en què les tasques creades executen la funció `doCompute`. D'aquesta manera l'arbre de recursivitat es va generant en paral·lel.



De nou fixem-nos que les tasques que acaben invocant a `doCompute` (les fulles de l'arbre de recursivitat) tenen la mínima granularitat (una invocació de `doCompute` de mida `MIN`). Si volem augmentar la seva granularitat el que podem fer és aturar la generació de tasques en nivells anteriors en l'arbre de recursivitat, per exemple:



Per exemple si el **cut-off ara no el fem per nivell sinó per mida de vector**, el codi seria tan fàcil com (per exemple amb `CUT-OFF` definit com a 16):

```
#define CUT_OFF 16

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

void main () {
   #pragma omp parallel
   #pragma omp single
   partition (vector, N);                      // N is multiple of P
}
```

Fixeu-vos que el condicional que controla el cas base de la recursivitat (`n>MIN`) i el condicional que controla el cut-off (`size>CUT_OFF`) són diferents. El primer decideix si arribem o no al cas base, i en cas negatiu, el segon controla si generem tasques o no.

Slide 29

En aquesta slide i següents veiem el suport que ens dóna OpenMP per implementar aquests mecanismes de cut-off. Bàsicament la clàusula `final` que afegida a `task` decideix si la tasca és final o no, és a dir, si a partir d'aquí s'han de generar més tasques o no. De fet la tasca es genera igualment, però l'executa la mateixa tasca pare. Per controlar la generació de tasques a partir d'aquest moment, OpenMP inclou una funció intrínseca que ens diu si estem en una tasca final o no, i així nosaltres decidir si generem tasca o no. El codi modificat fent ús d'aquesta clàusula i funció intrínseca seria:

```
void partition (int * vector, int n) {
   if (n > MIN) {                              // MIN is multiple of P
      int size = n / 4;
      if (!omp_in_final())
         for (int i = 0; i < 4; i++)
            #pragma omp task final(size<=CUT_OFF)
            partition (&vector[i*size], size);
      else
         for (int i = 0; i < 4; i++)
            partition (&vector[i*size], size);
   } else
      doComputation (vector, n);
}
```

Observeu que si la funció intrínseca ens retorna cert, l'execució se'n va per una branca on no generem tasques. En cas contrari, l'execució se'n va per l'altre branca on si generem tasques.

Slide 31

Per evitar aquesta transformació de codi OpenMP inclou una altra clàusula anomenada `mergeable`, el que passa és que no està implementada en els compiladors que tenim disponibles. Per tant no la farem servir, tot i que ens simplificaria molt la programació, tal com es mostra a continuació:

```
void partition (int * vector, int n) {
   if (n > MIN) {                              // MIN is multiple of P
      int size = n / 4;
      for (int i = 0; i < 4; i++)
         #pragma omp task final(size<=CUT_OFF) mergeable
         partition (&vector[i*size], size);
   } else
      doComputation (vector, n);
}
```

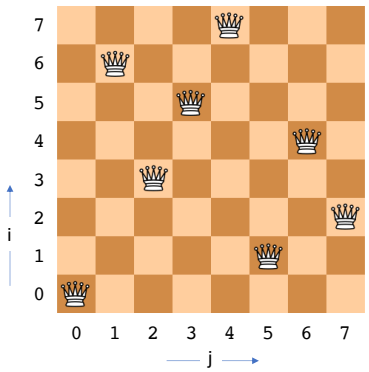
COMENTARI IMPORTANT: en les slides es fa servir un exemple diferent al fet servir en aquest guió. Interessant que us els mireu també de cara a estudiar el tema.

Slides 34 a 39

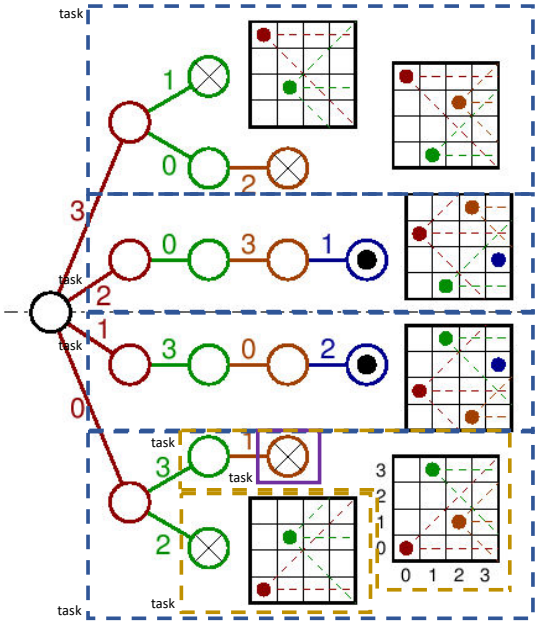
En aquestes slides teniu un altre exemple interessant d'algorisme recursiu fent ús de branch&bound per explorar configuracions vàlides pel problema de les "N Reines". Observeu que el tauler d'escacs es representa com un simple vector de "char" (8 bits) en el que cada posició representa una columna del tauler on s'indica la fila en què es troba la reina. Per exemple:

```
char *a;                        // Solution being explored
int  sol_count = 0;             // Total number of solutions found
int size = 8;                   // board size


void nqueens(int n, int j, char *a) {
     if (j == n) sol_count += 1;
     else
        // try each possible position for queen <j>
        for ( int i=0 ; i < n ; i++ ) {
            a[j] = (char) i;
            if (ok(j + 1, a)) nqueens(n, j + 1, a);
        }
}


int main() {
     a = alloca(size * sizeof(char));
     nqueens(size, 0, a);
}
```

a = [0, 6, 3, 5, 7, 1, 4, 2]



En la funció `nqueens` la variable `n` indica la mida del tauler i `j` indica la següent columna (`0..n-1`) a explorar. Quan `j==n` vol dir que ja tenim una solució completa. Si no, explorem per la nova columna totes les possibles posicions de la nova reina. La funció `ok` valida si la solució incompleta que es passa per l'argument és correcte (és a dir, és una configuració en què les reines col·locades no es maten entre elles). Si és vàlida, torna a cridar recursivament. Aquestes crides poden ser tasques encarregades d'explorar cadascuna una possibilitat diferent. La figura següent mostra l'arbre d'exploració pel cas de n=4, amb les tasques creades per algunes de les branques d'exploració.



Important observar, tal com mostra el codi de la slide 37, que cada tasca necessita un nou tauler per poder explorar les solucions que hi ha en la seva branca. Això ho fa la instrucció `alloca` (combinació de malloc i free, que "alocata" memòria al stack de la funció) que crea un nou tauler. Per què es necessita el `#pragma omp taskwait`? La tasca pare ha d'esperar que totes les tasques creades en aquest nivell acabin la seva exploració abans de "desalocatar" els seus taulers. Evidentment, l'increment del nombre de solucions trobades s'ha de fer evitant data races, en aquest cas fent ús de `atomic`.

En la slide 38 teniu les dues opcions per allocatar el nou tauler, amb `alloca` al stack o amb `malloc` al heap. `malloc` necessita fer el `free`

És important en aquest cas aplicar cut-off? Cada branca que s'explora requereix un nou tauler, a part de l'overhead de creació de la tasca que. Farà l'exploració. La slide 39 mostra el cut-off per nivell aplicat, fent ús de nou de les clàusules que proporciona OpenMP `final` i `omp_in_final`.

Finalment, observeu que l'estratègia leaf no té cap sentit en aquest codi, ja que no hi ha pràcticament computació en les fulles, només incrementar contador si `j==n`.

Problemes a fer: 1 (repàs de iterative), 3 (repàs de leaf vs. tree), 6 (escriure codis paral·lels leaf/tree amb cut-off)