

PAR TASK DECOMPOSITION STRAT.

Control de generación de tareas:

Iterativas:

Se controla la granularidad de las tareas config. el #iteraciones ejecutadas x cada tarea

Recursivas

Se controla la granularidad de las tareas controlando los niveles de recursión donde las tareas se generan → CUT-OFF

se puede controlar de manera

- estática: después de x llamadas.
- dinámica: en función de las tareas.

Iterative Task Decomposition Countable

La granularidad se define con el #iteraciones.

Ejemplo tareas implícitas

```
void vector_add(int *A, int *B, int *C, int n) {  
    int who = omp_get_thread_num();  
    int nt = omp_get_num_threads();  
    int BS = n / nt;  
    for (int i = who*BS; i < (who+1)*BS; i++)  
        C[i] = A[i] + B[i];  
}  
  
void main() {  
    ...  
    #pragma omp parallel  
    vector_add(a, b, c, N);  
    ...  
}
```

Cada tarea implícita ejecuta un subconjunto de iteraciones.

Ejemplo tareas explícitas:

```
void vector_add(int *A, int *B, int *C, int n) {  
    for (int i=0; i< n; i++)  
        #pragma omp task  
        C[i] = A[i] + B[i];  
}  
  
void main() {  
    ...  
    #pragma omp parallel  
    #pragma omp single  
    vector_add(a, b, c, N);  
    ...  
}
```

Cada tarea explícita ejecuta 1 iteración del i loop.

→ Granularidad DEMASIADO fina, overhead.

Ejemplo chunk de BS iteraciones (explícitas)

```
void vector_add(int *A, int *B, int *C, int n) {
    int BS = ...
    for (int ii=0; ii< n; ii+=BS)
        #pragma omp task
        for (int i = ii; i < min(ii+BS, n); i++)
            C[i] = A[i] + B[i];
}
void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    vector_add(a, b, c, N);
    ...
}
```

Mejora de la opción anterior.

Ejemplo taskloop (explícitas)

```
void vector_add(int *A, int *B, int *C, int n) {
    int BS = ...
    #pragma omp taskloop grainsize(BS)      // or alternatively num_tasks(n/BS)
    for (int i=0; i< n; i++)
        C[i] = A[i] + B[i];
}
void main() {
    #pragma omp parallel
    #pragma omp single
    ... vector_add(a, b, c, N); ...
}
```

Iterative Task Decomposition Uncountable loop:

```
int main() {
    struct node *p;
    p = init_list(n);
    ...
    #pragma omp parallel
    #pragma omp single
    while (p != NULL) {
        #pragma omp task firstprivate(p) // see note below
        process_work(p);
        p = p->next;
    }
    ...
}
```

Granularidad: 1 iteración
 Objetivo: amortizar el Creation Overhead

Recursive Task Decomposition Leaf Strategy → CASO BASE

La tarea corresponde con cada invocación de la función que se llama cuando acaban las llamadas recursivas.

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)
        result += A[i] * B[i];
}

void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    } else
        #pragma omp task
        dot_product(A, B, n);
}

void main() {
    #pragma omp parallel
    #pragma omp single
    rec_dot_product(a, b, N);
}
```

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)
        result += A[i] * B[i];
}

void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    } else
        #pragma omp task
        dot_product(A, B, n);
}
```

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++)
        tmp += A[i] * B[i];
    #pragma omp atomic
    result += tmp;
}

void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    } else
        #pragma omp task
        dot_product(A, B, n);
}
```

Leaf Strategy Depth Control:

```
#define CUTOFF 2
...
void rec_dot_product(int *A, int *B, int n, int depth) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth == CUTOFF)
            #pragma omp task
            {
                rec_dot_product(A, B, n2, depth+1);
                rec_dot_product(A+n2, B+n2, n-n2, depth+1);
            }
        else {
            rec_dot_product(A, B, n2, depth+1);
            rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    }
    else // if recursion finished, need to check if task has been generated
        if (depth <= CUTOFF)
            #pragma omp task
            dot_product(A, B, n);
        else
            dot_product(A, B, n);
}
...

```

Tree Strategy

La tarea corresponde con cada invocación de la llamada recursiva.

```
int dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++) tmp += A[i] * B[i];
    return(tmp);
}

int rec_dot_product(int *A, int *B, int n) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        #pragma omp task shared(tmp1) // firstprivate(A, B, n, n2) by default
        tmp1 = rec_dot_product(A, B, n2);
        #pragma omp task shared(tmp2) // firstprivate(A, B, n, n2) by default
        tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
        #pragma omp taskwait
    } else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}

void main() {
    #pragma omp parallel
    #pragma omp single
    result = rec_dot_product(a, b, N);
}
```

Tree Strategy Depth Control:

```
#define N 1024
#define MIN_SIZE 64
#define CUTOFF 3

int rec_dot_product(int *A, int *B, int n, int depth) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth < CUTOFF) {
            #pragma omp task shared(tmp1)
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            #pragma omp task shared(tmp2)
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
            #pragma omp taskwait
        } else {
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    }
    else tmp = dot_product(A, B, n);
    return(tmp1+tmp2);
}
```

OpenMP support for CUT-OFF

Final: si la expresión de la cláusula evalúa true, la tarea generada & sus descendientes, serán FINALES.

omp_in_final: devuelve true si está en una región de una tarea final.

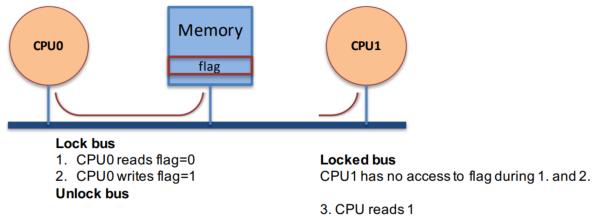
```
#define MIN_SIZE 64
#define CUTOFF 3
...
int rec_dot_product(int *A, int *B, int n, int depth) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (!omp_in_final()) {
            #pragma omp task shared(tmp1) final(depth >= CUTOFF)
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            #pragma omp task shared(tmp2) final(depth >= CUTOFF)
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
            #pragma omp taskwait
        } else {
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    } else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}
...
```

N-QUEENS EXAMPLE

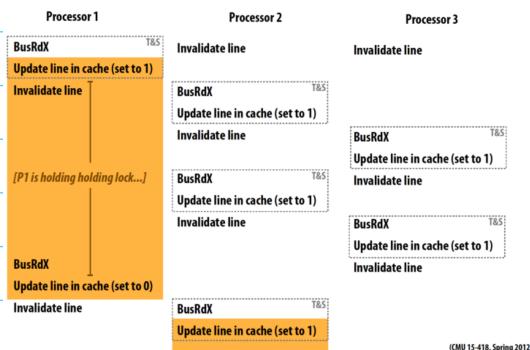
```
void nqueens(int n, int j, char *a) {
    if (j == n)
        #pragma omp atomic
        sol_count += 1;
    else
        // try each possible position for queen <j>
        if (!omp_in_final())
            for ( int i=0 ; i < n ; i++ ) {
                a[j] = (char) i;
                if (ok(j + 1, a))
                    // allocate a temporary array and copy <a> into it
                    char * b = alloca(n * sizeof(char));
                    memcpy(b, a, (j + 1) * sizeof(char));
                    #pragma omp task final(j>CUT_OFF)
                    nqueens(n, j + 1, b);
            }
            #pragma omp taskwait
        } else
            for ( int i=0 ; i < n ; i++ ) {
                a[j] = (char) i;
                if (ok(j + 1, a)) nqueens(n, j + 1, a);
            }
}
```

Sincronización a bajo nivel:

Hardware: necesario soporte hw para garantizar instrucciones atómicas (indivisibles) para hacer fetch y actualizar mem.



• Test-and-set: leer value y settearlo a 1.



```
set_lock: t&s r2, flag  
          bneq r2, set_lock      // already locked?  
          ...  
unset_lock: st flag, #0      // free lock
```

• Atomic exchange: intercambio de un valor en un registro y un valor en memoria.

```
set_lock: exch r2, flag      // atomic exchange  
          bneq r2, set_lock      // already locked?  
          ...  
unset_lock: st flag, #0      // free lock
```

• Fetch-and-op: leer un valor y reemplazarlo con un resultado después de una operación aritmética simple (atomic)

• Test-test-and-set: reduce el ancho de banda de memoria y las operaciones de concurrencia necesarias requeridas por test-and-set.

```
set_lock: ld r2, flag        // test with regular load  
          bneq r2, set_lock    // lock is cached meanwhile it is not updated  
          t&s r2, flag        // test if the lock is free  
          bneq r2, set_lock    // test and acquire lock if STILL free  
          ...  
unset_lock: st flag, #0      // free the lock
```

Alternativa para sistemas grandes (atomicidad)

Load-linked-Store Conditional (ll-sc)

- LL: valor actual de una @mem.
- Sc: guarda un nuevo valor en @mem si no se ha actualizado antes. Devuelve 1 ✓ o Ø ✗

difícil o
ineficiente)

```
// exchange r4 with location.  
try: mov r3, r4  
      ll r2, location  
      sc r3, location  
      beqz r3, try  
      mov r4, r2
```

```
// add 1 to location  
try: ll r2, location  
      add r3, r2, #1  
      sc r3, location  
      beqz r3, try
```

```
set_lock: ll r2, flag          // first test with load linked  
                                // lock is cached meanwhile it is not updated  
bnez r2, set_lock            // test if the lock is free  
mov r2, #1  
sc r2, flag                  // try to store 1  
beqz r2, set_lock            // repeat if someone else did it before me  
...  
unset_lock: st flag, #0        // free the lock
```