Parallelism (PAR)

Parallel programming principles: Task decomposition

Eduard Ayguadé, José Ramón Herrero and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2019/20 (Fall semester)

Additional learning material for this lesson

- Atenea: Unit 3 Task decomposition
  - Video lesson 4 (introducing/complementing slides 5 and 8–24)
  - Questions after video lesson 4
  - Going further: cut-off based on number of tasks pending to be executed (OPTIONAL)
- Collection of Exercises: problems in Chapter 3

Objective: problem decomposition

- From a specification that solves the original problem, find a decomposition of the problem
  - Identify pieces of work (tasks) that can be done concurrently
  - Identify data structures (input, output and/or intermediate) or parts of them that can be managed concurrently

  ... ensuring that the same result is produced
  - Identify dependencies that impose ordering constraints (synchronizations) and data sharing

Objective: problem decomposition

- Having in mind two productivity goals:
  - Performance: maximize concurrency and reduce overheads (maximize potential speedup)
  - Programmability: readability and portability, target architecture independency
- Two usual approaches to problem decomposition
  - Task decomposition (Chapter 3)
  - Data decomposition (Chapter 5)

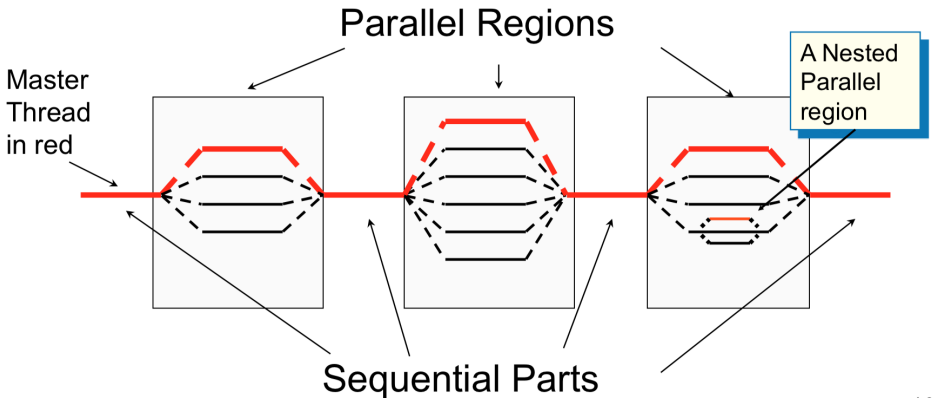
  Either way, you'll look at both tasks and data; difference is in which you look at first, and then the other follows

Identifying tasks in your sequential program (patterns)

- Linear task decomposition
  - Task = code block or procedure invocation
- (Linear) Iterative task decomposition
  - Tasks = body of iterative constructs, such as loops (countable or uncountable)
  - Examples: Pi computation, Mandelbrot and heat diffusion equation in lab sessions, vector and matrix operations, ...
- Recursive task decompostition
  - Tasks = recursive procedure invocations, for example in divide–and–conquer problems
  - Examples: Fibonacci, multisort in lab session, branch and bound problems, ...

Task creation in OpenMP (summary)

- `#pragma omp parallel`: One **implicit** task is created for each thread in the team (and immediately executed)



- `#pragma omp task`: One **explicit** task is created, packaging code and data for (possible) deferred execution
  - Tasks executed by threads in the `parallel` region

Outline

Example 1: iterative sum of two vectors

Simple example to show how to use implicit and explicit tasks to express the available parallelism: all iterations of the loop are independent

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

void main() {
   ....
   vector_add(a, b, c, N);
   ...
}
```

Implicit tasks: how to share work? (1)

(Linear) iterative task decomposition using:

- Manual work distribution, one possibility could be:

```
void vector_add(int *A, int *B, int *C, int n) {
   int nt = omp_get_num_threads();
   int who = omp_get_thread_num();
   for (int i = who; i< n; i += nt)
       C[i] = A[i] + B[i];
}

void main() {
   ....
   #pragma omp parallel
   vector_add(a, b, c, N);
   ...
}
```

Each implicit task executes a subset of iterations from the loop that is determined by the thread identifier executing the implicit task (`omp_get_thread_num()`) and the total number of implicit tasks (or equivalently, number of threads in the team `omp_get_num_threads()`).

Implicit tasks: how to share work? (2)

(Linear) iterative task decomposition using:

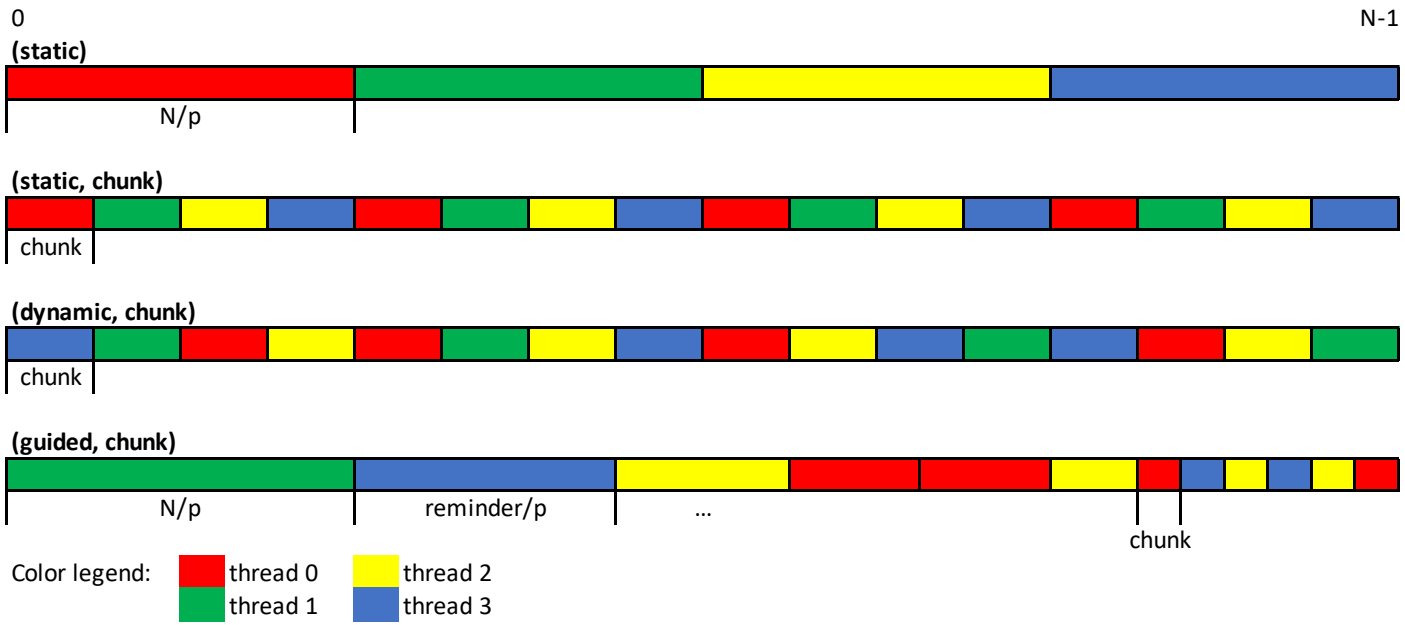
- `for` work–sharing construct

```
void vector_add(int *A, int *B, int *C, int n) {
   #pragma omp for schedule(static)
   for (int i=0; i< n; i++)
       C[i] = A[i] + B[i];
}

void main() {
   ....
   #pragma omp parallel
   vector_add(a, b, c, N);
   ...
}
```

Each implicit task executes chunks of iterations, depending on what is specified in the `schedule` clause. Implicit barrier at the end of each work–sharing (`nowait` clause to skip it).

`Schedule` clause in `for` work–sharing (summary)

Different options to assign chunks of iterations to each implicit task through the `schedule` clause



Implicit tasks: how to share work? (3)

Work–sharing in nested loops with `collapse` clause

```
#pragma omp for collapse(2) schedule(static, 2)
for ( i=0; i<N; i++ )
    for ( j=0; j<N; j++ )
        A[i][j] = B[i][j] + C[i][j]
```

- Useful when there is insufficient work in one loop
- It is only possible to collapse perfectly nested loops (i.e. no code in between loops, no triangular loops, ...).
- Iterations are scheduled as if their execution is linearized, in this example for $N = 4$

(i, j)	(0, 0)	(0, 1)	(0, 2)	(0, 3)	(1, 0)	(1, 1)	(1, 2)	(1, 3)	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(3, 0)	(3, 1)	(3, 2)	(3, 3)
	thread 0		thread 1		thread 2		thread 3		thread 0		thread 1		thread 2		thread 3	

Outline

Explicit tasks: linear task decomposition

E.g. a task is a sequence of instructions, as for example in sum of two vectors artificially divided in two parts:

```
void vector_add(int *A, int *B, int *C, int n) {
   #pragma omp task
   for (int i=0; i< n/2; i++)
       C[i] = A[i] + B[i];

   #pragma omp task
   for (int i=n/2; i< n; i++)
       C[i] = A[i] + B[i];
}

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

One task generator (use of `single` work–sharing).

Explicit tasks: (linear) iterative task decomposition (1)

But more naturally, a task can be a single loop iteration:

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; ii++)
       #pragma omp task
       C[i] = A[i] + B[i];
}

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

Each explicit task executes a single iteration of the `i` loop, large task creation overhead, very fine granularity!

Explicit tasks: (linear) iterative task decomposition (2)

Chunk of loop iterations, requires loop transformation:

```
void vector_add(int *A, int *B, int *C, int n) {
   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);
   ...
}
```

Outer loop jumps over chunks of BS iterations, inner loop traverses each chunk

Explicit tasks: (linear) iterative task decomposition (3)

OpenMP provides an alternative construct to specify tasks out of loop iterations:

```
void vector_add(int *A, int *B, int *C, int n) {
   #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); ...
}
```

- `grainsize(m)`: each task executes $[min(m, n) \; .. \; 2 \times m]$ consecutive iterations, being n the total number of iterations
- `num_tasks(m)`: creates as many tasks as $min(m, n)$

Explicit tasks: (linear) iterative task decomposition (4)

List of elements, traversed using an uncountable (while) loop

```
int main() {
struct node *p;

p = init_list(n);
...

#pragma omp parallel
#pragma omp single
while (p != NULL) {
    #pragma omp task firstprivate(p)
    process_work(p);
    p = p->next;
    }
...
}
```

Need to capture the value of p at task creation time to allow its (possibly) deferred execution (`firstprivate` clause).

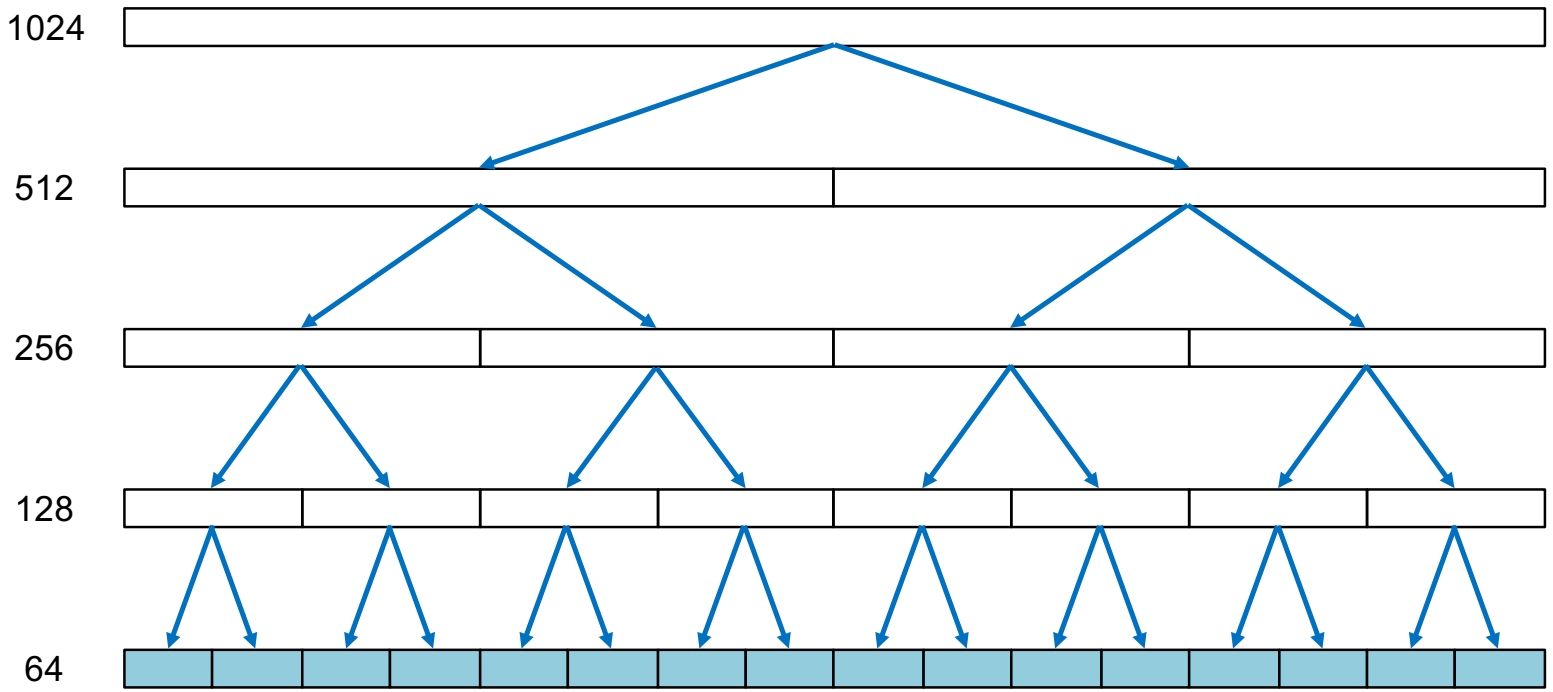
Example 2: recursive sum of two vectors

"Divide–and–conquer" strategy: recursively divide the problem into smaller sub-problems

```
#define N 1024
#define MIN_SIZE 64

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

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       rec_vector_add(A, B, C, n2);
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   }
   else vector_add(A, B, C, n);
}
void main() {
   ....
   rec_vector_add(a, b, c, N);
   ...
}
```

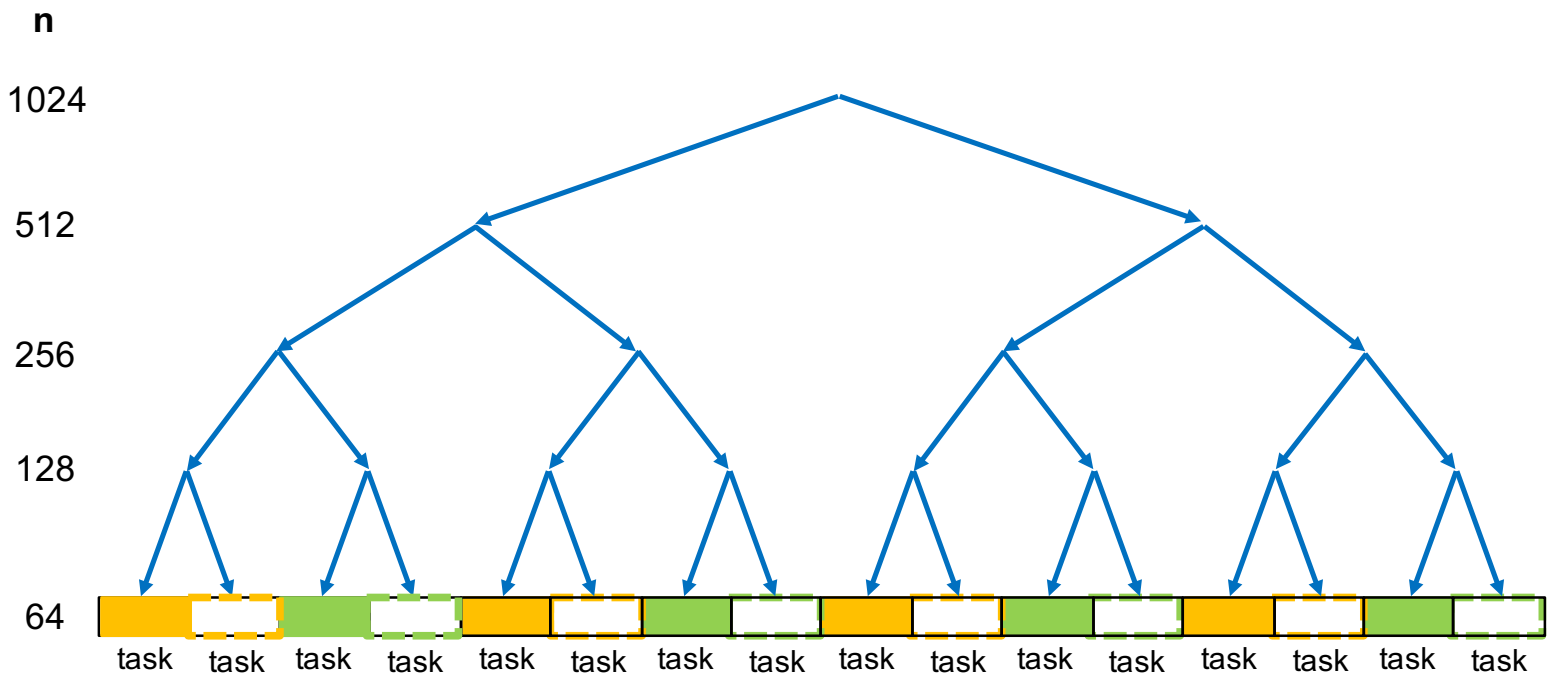
Example 2: recursive sum of two vectors

```
N=1024, MIN_SIZE=64
```



Explicit tasks: recursive task decomposition strategies

Leaf strategy: a task corresponds with each invocation of `vector_add` once the recursive invocations stop



- Sequential generation of tasks

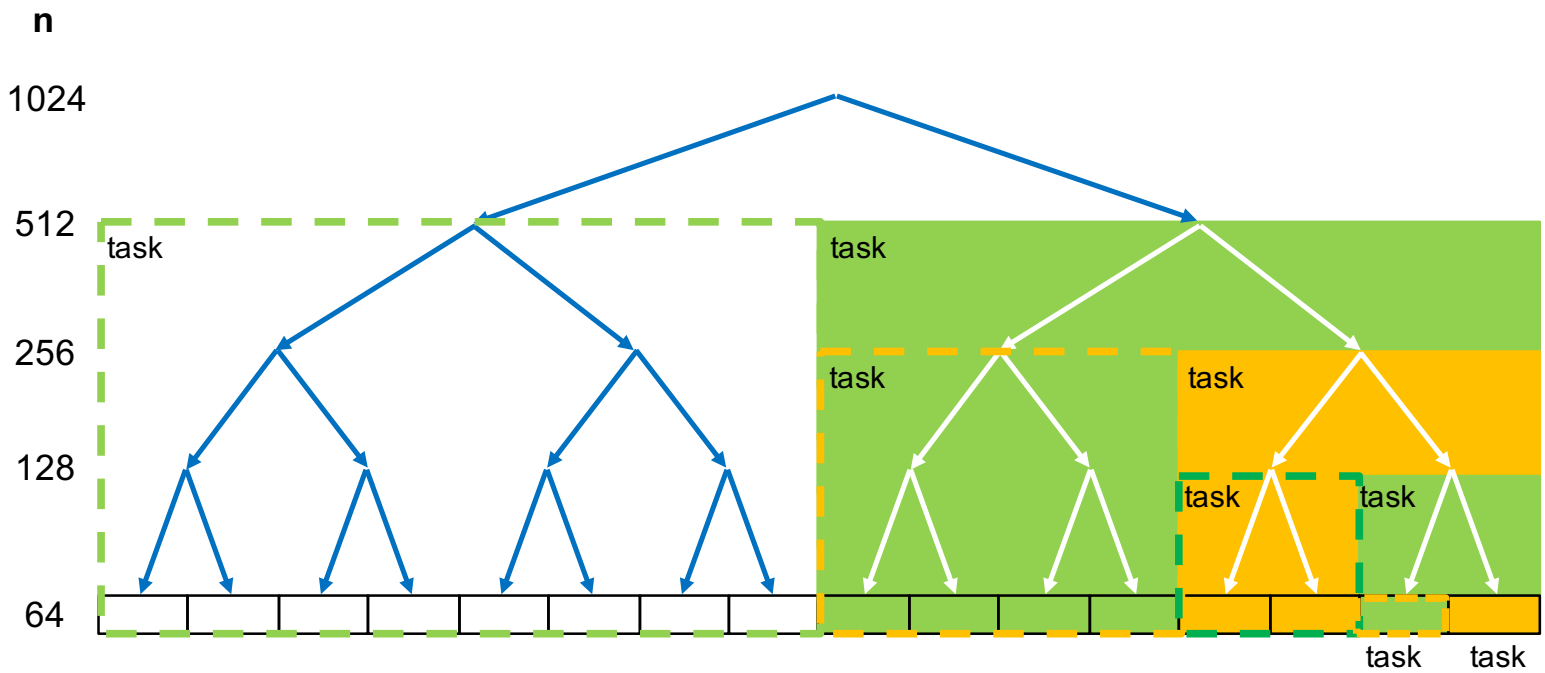
Implementing the decomposition strategies

Leaf parallelization

```
#define N 1024
#define MIN_SIZE 64

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

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

Explicit tasks: recursive task decomposition strategies

Tree strategy: a task corresponds with each invocation of `rec_vector_add`



- Parallel generation of tasks
- Granularity: some tasks simply generate new tasks

Implementing the decomposition strategies (cont.)

Tree parallelization

```
#define N 1024
#define MIN_SIZE 64

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

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

How would you address the N-queens problem?

```
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;

     // 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);
}
```

How would you address the N-queens problem? (cont.)

```
void nqueens(int n, int j, char *a) {
     if (j == n)
          #pragma omp atomic
          sol_count += 1;

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

int main() {
     a = alloca(size * sizeof(char));
     #pragma omp parallel
     #pragma omp single
     nqueens(size, 0, a);
}
```

A new board for each task is needed, why?

Outline

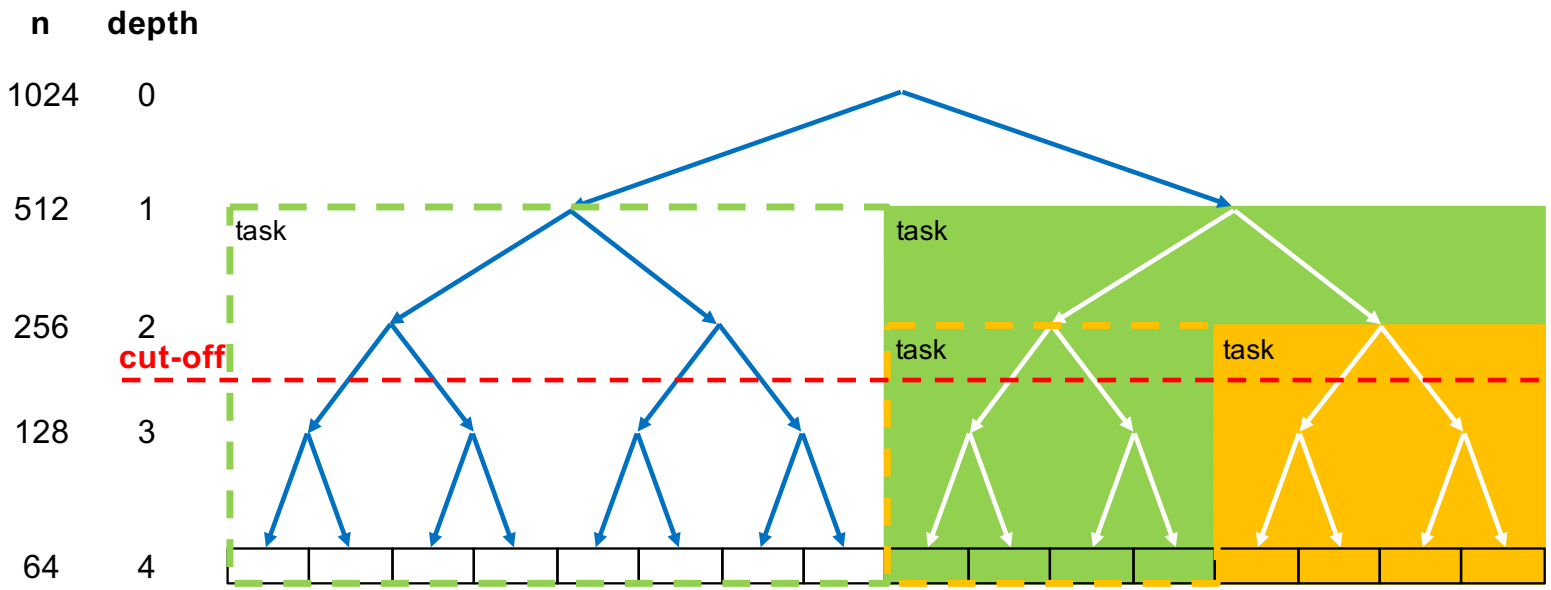
Task generation control

Excessive task generation may not be necessary (i.e. cause excessive overhead)

- In iterative `taskloop` decompositions one can easily control task granularity (either using `grainsize` or `num_tasks`).
- In recursive `task` decompositions one needs to stop task generation ... **cut-off control**
  - after certain number of recursive calls (static control)
  - when the size of the vector is too small (static control)
  - when the number of generated tasks is too much (dynamic control)
  - ...

Cut-off control

Tree parallelization with **depth recursion control**



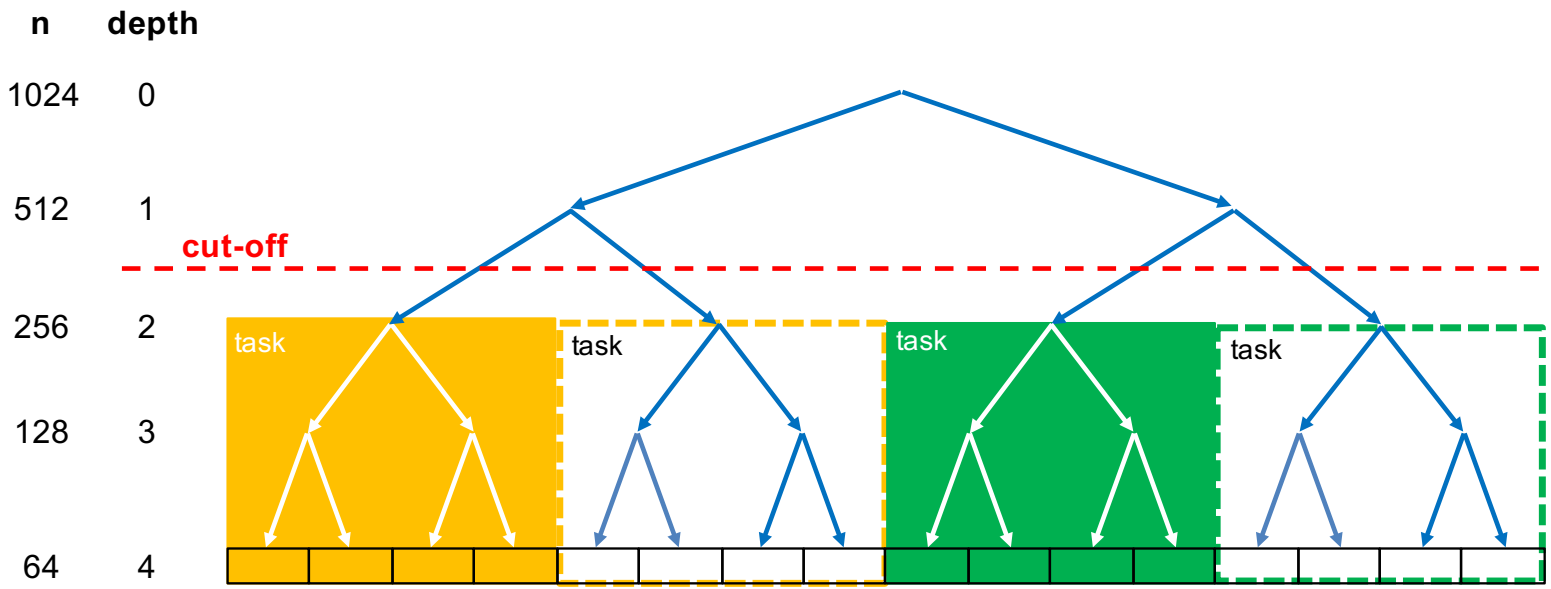
Cut-off control (cont.)

Tree parallelization with **depth recursion control**

```
#define CUTOFF 3
...
void rec_vector_add(int *A, int *B, int *C, int n, int depth) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth < CUTOFF) {
            #pragma omp task
            rec_vector_add(A, B, C, n2, depth+1);
            #pragma omp task
            rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
        }
        else {
            rec_vector_add(A, B, C, n2, depth+1);
            rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
        }
    } else vector_add(A, B, C, n);
}

void main() {
    ....
    #pragma omp parallel
    #pragma omp single
    rec_vector_add(a, b, c, N, 0);
    ...
}
```

Cut-off control

Leaf parallelization with **depth recursion control**



Cut-off control (cont.)

Leaf parallelization with **depth recursion control**

```
#define CUTOFF 2
...
void rec_vector_add(int *A, int *B, int *C, int n, int depth) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth == CUTOFF) {
            #pragma omp task
            rec_vector_add(A, B, C, n2, depth+1);
            #pragma omp task
            rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
        }
        else {
            rec_vector_add(A, B, C, n2, depth+1);
            rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
        }
    }
    else
        if (depth <= CUTOFF)
            #pragma omp task
            vector_add(A, B, C, n);
        else
            vector_add(A, B, C, n);
}
...
```

OpenMP support for cut-off

- `final` clause: If the expression of a `final` clause evaluates to *true* the generated task and **all of its descendent tasks** will be final. The execution of a final task is sequentially **included** in the generating task (but the task is still generated)

- `omp_in_final()` intrinsic function: it returns true when executed in a final task region; otherwise, it returns false.

Very simple example with cut-off (rewritten)

Making use of `omp_in_final`:

```
#define MIN_SIZE 64
#define CUTOFF 3
...
void rec_vector_add(int *A, int *B, int *C, int n, int depth) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (!omp_in_final()) {
            #pragma omp task final(depth >= CUTOFF)
            rec_vector_add(A, B, C, n2, depth+1);
            #pragma omp task final(depth >= CUTOFF)
            rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
        }
        else {
            rec_vector_add(A, B, C, n2, depth+1);
            rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
        }
    }
   else vector_add(A, B, C, n);
}
...
```

OpenMP support for cut-off (cont.)

- `mergeable` clause: when a mergeable clause is present on a task construct, and the generated task is an undeferred task or an included task, then the compiler may choose to generate a merged task instead. If a merged task is generated, then the behavior is as though there was no task directive at all
  - Not implemented in all compilers, so the optimization needs to be implemented by the programmer using the `omp_in_final` instrinsic

Very simple example with cut-off (rewritten once more)

```
#define MIN_SIZE 64
#define CUTOFF 3

void rec_vector_add(int *A, int *B, int *C, int n, int depth) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       #pragma omp task final(depth >= CUTOFF) mergeable
       rec_vector_add(A, B, C, n2, depth+1);
       #pragma omp task final(depth >= CUTOFF) mergeable
       rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
   }
   else vector_add(A, B, C, n);
}
...
```

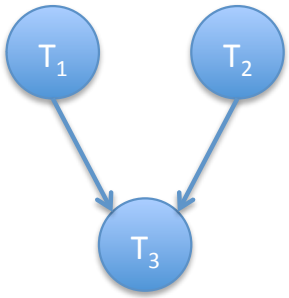
Adding cut-off to N-queens (recursion level)

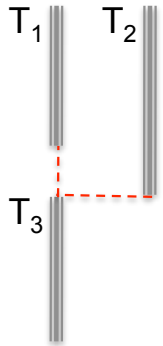
```
void nqueens(int n, int j, char *a) {
     if (j == n)
          #pragma omp atomic
          sol_count += 1;

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

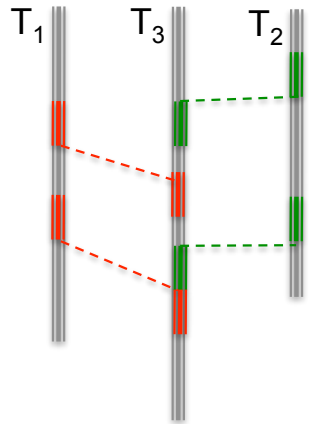
Outline

Dependences

- Constraints in the parallel execution of tasks
  - Task ordering constraints: they force the execution of (groups of) tasks in a required order
  - Data sharing constraints: they force the access to data to fulfil certain properties (write-after-read, exclusive, commutative, ...)







Dependences (cont.)

- If no constraints are defined during the parallel execution of the tasks, the algorithm is called "embarrassingly" parallel. There are still challenges in this case:
  - Load balancing: how to distribute the work to perform so the load is evenly balanced among tasks (e.g. pi computation vs. Mandelbrot set)
  - Data locality: how to assign work to tasks so that data resides in the memory hierarchy levels close to the processor

Protecting task interactions

Identify the sequence of statements in a task that may conflict with a sequence of statements in another task, creating a possible data race

- Conflict exists if both sequences access the same data and at least one of them modifies the data

Two mechanisms:

- Atomic accesses: mechanism to guarantee atomicity in load/store instructions
- Mutual exclusion: mechanism to ensure that only one task at a time executes the code within a critical section

Task interactions: atomic access to variables

```
#pragma omp atomic [update | read | write]
     expression
```

- Ensures that a specific storage location is accessed atomically, avoiding the possibility of multiple, simultaneous reading and writing threads
  - Atomic updates: `x += 1`, `x = x - foo()`, `x[index[i]]++`
  - Atomic reads: `value = *p`
  - Atomic writes: `*p = value`
- Only protects the read/operation/write
- Usually more efficient than a `critical` construct

Example 3: iterative dot product

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

We need to protect each update of the shared variable `result`.
Too much data sharing overhead per task!

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

Example 4: recursive dot product

```
#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++)
       #pragma omp atomic
       result += A[i] * B[i];
}

void rec_dot_product(int *A, int *B, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       #pragma omp task // firstprivate(n, n2) by default
       rec_dot_product(A, B, n2);
       #pragma omp task // firstprivate(n, n2) by default
       rec_dot_product(A+n2, B+n2, n-n2);
   }
   else dot_product(A, B, n);
}
void main() {
   #pragma omp parallel
   #pragma omp single
   rec_dot_product(a, b, N);
}
```

Example 4: recursive dot product (cont.)

Each individual update of the shared variable `result` causes too much overhead, how to reduce it?

```
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;
}
```

Task interactions: mutual exclusion

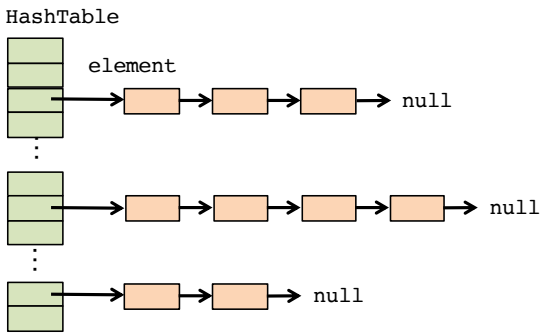
In OpenMP there exist two alternatives for mutual exclusion: `critical` and low-level synchronisation functions (locks)

Also, two options for mutual exclusion using `critical` regions:

- `critical` pragma: a thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program)
- `critical(name)` pragma: the `name` allows the programmer to differentiate disjoint sets of critical sections (name is a label, not a program variable)

Example 5: inserting elements in hash table

- Hash table defined as a collection of linked lists




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

element * HashTable[SIZE_TABLE];

for (i = 0; i < elements; i++) {
        index = hash_function(element[i]);
        insert element (element[i], index);
}
```

- Updates to the list in any particular slot must be protected to prevent a race condition

```
#pragma omp parallel for private (index)
for (i = 0; i < elements; i++) {
    index = hash_function (element[i]);
    #pragma omp critical  // atomic not possible here
    insert_element (element[i], index);
}
```

Task interactions: mutual exclusion

Low-level synchronization functions using *locks*: special variables that live in memory with two basic operations:

- Acquire: while a thread has the lock, nobody else gets it; this allows the thread to do its work in private, not bothered by other threads
- Release: allow other threads to acquire the lock and do their work (one at a time) in private
- Type definition and instrinsics:

```
void omp_init_lock(omp_lock_t *lock)
void omp_destroy_lock(omp_lock_t *lock)

void omp_set_lock(omp_lock_t *lock)
void omp_unset_lock(omp_lock_t *lock)

int omp_test_lock(omp_lock_t *lock)
```

Example 5: inserting elements in hash table (cont.)

Associate a lock variable with each slot in the hash table, protecting the chain of elements in an slot

```
omp_lock_t hash_lock[HASH_TABLE_SIZE];

for (i = 0; i < HASH_TABLE_SIZE; i++)
    omp_init_lock(&hash_lock[i]);

#pragma omp parallel for private (index)
for (i = 0; i < elements; i++) {
    index = hash_function (element[i]);
    omp_set_lock (&hash_lock[index]);
    insert_element (element[i], index);
    omp_unset_lock (&hash_lock[index]);
}

for (i = 0; i < HASH_TABLE_SIZE; i++)
    omp_destroy_lock(&hash_lock[i]);
```

Threads may be inserting elements into the hash table in parallel, as long as these elements hash to different slots

Reducing task interactions

Task interactions introduce overheads. How can one reduce them?

- Separable dependencies (reductions): the dependencies between tasks can be managed by replicating key data structures and locally working with these local structures. When appropriate, locally replicated data structures are combined into the final global result (task interaction)

```
double max_val=0.0;

#pragma omp parallel for reduction(max : max_val)
for( i = 0; i < N; i++) {
    if(vector[i] > max_val)
        max_val = vector[i];
}
```

Outline

Task ordering constraints

- Two possible sources of constraints:
  - Control flow constraints: the creation of a task depends on the outcome (decision) of one or more previous tasks
  - Data flow constraints: the execution of a task can not start until one or more previous tasks have computed/used some data
    - *read–after–write*, *write–after–read* and *write–after–write*
- Task ordering constraints are easily imposed by sequentially composing tasks and/or using global synchronizations.

Task synchronization in OpenMP (summary)

- Thread barriers: wait for all threads to finish previous work (`#pragma omp barrier` and all implicit barriers at the end of work–sharing constructs)
- Task barriers:
  - `taskwait`: Suspends the current task waiting on the completion of **child tasks** of the current task. The taskwait construct is a stand-alone directive.
  - `taskgroup`: Suspends the current task at the end of structured block waiting on completion of **child tasks** of the current task **and their descendent** tasks.
- Task dependences

taskwait vs. taskgroup

```
#pragma omp task {}      // T1
#pragma omp task         // T2
{
    #pragma omp task {}  // T3
}
#pragma omp task {}      // T4

#pragma omp taskwait
// Only T1, T2 and T4 are guaranteed to have finished at this point
```

```
#pragma omp task {}          // T1
#pragma omp taskgroup
{
    #pragma omp task         // T2
    {
        #pragma omp task {}  // T3
    }
    #pragma omp task {}      // T4
}
// Only T2, T3 and T4 are guaranteed to have finished at this point
```

Example 4 revisited: recursive dot product

```
#define N 1024
#define MIN_SIZE 64

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 tmp = 0;
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       tmp += rec_dot_product(A, B, n2);
       tmp += rec_dot_product(A+n2, B+n2, n-n2);
   }
   else tmp = dot_product(A, B, n);
   return(tmp);
}
void main() {
   ....
   result = rec_dot_product(a, b, N);
   ...
}
```

Example 4 revisited: recursive dot product (cont.)

```
#define N 1024
#define MIN_SIZE 64

int rec_dot_product(int *A, int *B, int n) {
   int tmp1 = 0, tmp2 = 0;
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       #pragma omp task shared(tmp1) // firstprivate(n, n2) by default
       tmp1 = rec_dot_product(A, B, n2);
       #pragma omp task shared(tmp2) // firstprivate(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);
    ...
}
```

- `taskwait` guarantees that both instances of `rec_dot_product` finish before returning the result

Task dependences

- OpenMP allows the specification of dependences (through argument directionality) between sibling tasks (i.e. from the same parent task)

```
#pragma omp task [depend (in : var_list)]
                 [depend (out : var_list)]
                 [depend (inout : var_list)]
```

Task dependences are derived from the dependence type (`in`, `out` or `inout`) and its items in `var_list`. This list may include array sections

Task dependences

- The `in` dependence-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` dependence-type list
- The `out` and `inout` dependence-types: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` dependence-type list.

Example 6: wavefront execution with task dependences

- Function `foo(i, j)` processes *block(i, j)*
- Wave-front execution: the execution of `foo(i,j)` depends on `foo(i-1, j)` and `foo(i, j-1)`

```
#pragma omp parallel private(i, j)
#pragma omp single
{
   for (i=1; i<n i++) {
        for (j=1; j<n;j++) {
           #pragma omp task // firstprivate(i, j) by default
                            depend(in : block[i-1][j], block[i][j-1])
                            depend(out: block[i][j])
           foo(i,j);
         }
    }
}
```

Ordered execution of `for` work–sharing construct

A **doacross** loop is a loop nest where cross-iteration dependences exist

- The `ordered(n)` clause with an integer argument `n` is used to define the number of loops within the doacross nest
- `depend` clauses on `ordered` constructs within an `ordered` loop describe the dependences of the doacross loops
  - `depend(sink:expr)` defines the wait point for the completion of computation in a previous iteration defined by `expr`
  - `depend(source)` indicates the completion of computation from the current iteration

The doacross loop nest: first example

```
#pragma omp for schedule(static, 1) ordered(1)
for ( i = 1; i < N; i++ ) {
      A[i] = foo (i);
      #pragma omp ordered depend(sink: i-1)
      B[i] = goo( A[i], B[i-1] );
      #pragma omp ordered depend(source)
      C[i] = too( B[i] );
      }
```

- In this example an `i-1` to `i` cross-iteration dependence is defined, but only for the statement computing `B`.
- The computation of `A` and `C` can go in parallel across multiple iterations.

The doacross loop nest: wavefront example revisited

- Function `foo(i, j)` processes *block(i, j)*
- Wave-front execution: the execution of `foo(i,j)` depends on `foo(i-1, j)` and `foo(i, j-1)`

```
#pragma omp for schedule(static,1) ordered(2)
for ( i = 1; i < N; i++ )
   for ( j = 1; j < N; j++ ) {
      #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
      foo (i, j);
      #pragma omp ordered depend(source)
      }
```

The doacross loop nest: last example

```
#pragma omp for collapse(2) ordered(2)
for (i = 1; i < N-1; i++) {
  for (j = 1; j < M-1; j++) {
    A[i][j] = foo(i, j);
    #pragma omp ordered depend(source)
    B[i][j] = alpha * A[i][j];
    #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
    C[i][j] = 0.2 * (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]);
    }
}
```

- In this example the `i` and `j` loops are the associated loops for the collapsed loop as well as for the doacross loop nest.
- Note that in this case the dependence `source` directive is placed before the corresponding `sink` directive.

Parallelism (PAR)

Parallel programming principles: Task decomposition

Eduard Ayguadé, José Ramón Herrero and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2019/20 (Fall semester)