Parallelism (PAR)

Parallel programming principles: Task decomposition

Eduard Ayguadé and José Ramón Herrero

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2017/18 (Fall semester)

Outline

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

Outline

Task creation in OpenMP (summary)

- `#pragma omp parallel`: One **implicit** task is created for each thread in the team (and immediately executed)
  - `#pragma omp for` worksharing: iterations in a loop are distributed among the implicit tasks in the `parallel` region
- `#pragma omp task`: One **explicit** task is created, packaging code and data for (possible) deferred execution
  - Tasks can be nested

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, ...

Example 1: linear task decomposition

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);
   ...
}
```

Example 2: (linear) iterative task decomposition

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!

Example 2: (linear) iterative task decomposition (cont.)

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

Example 2: (linear) iterative task decomposition (cont.)

OpenMP provides two alternative constructs, one using `taskloop`:

```
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); ...
}
```

and another using implicit tasks and worksharing:

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

Example 3: (linear) iterative task decomposition

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;
    }
...
}
```

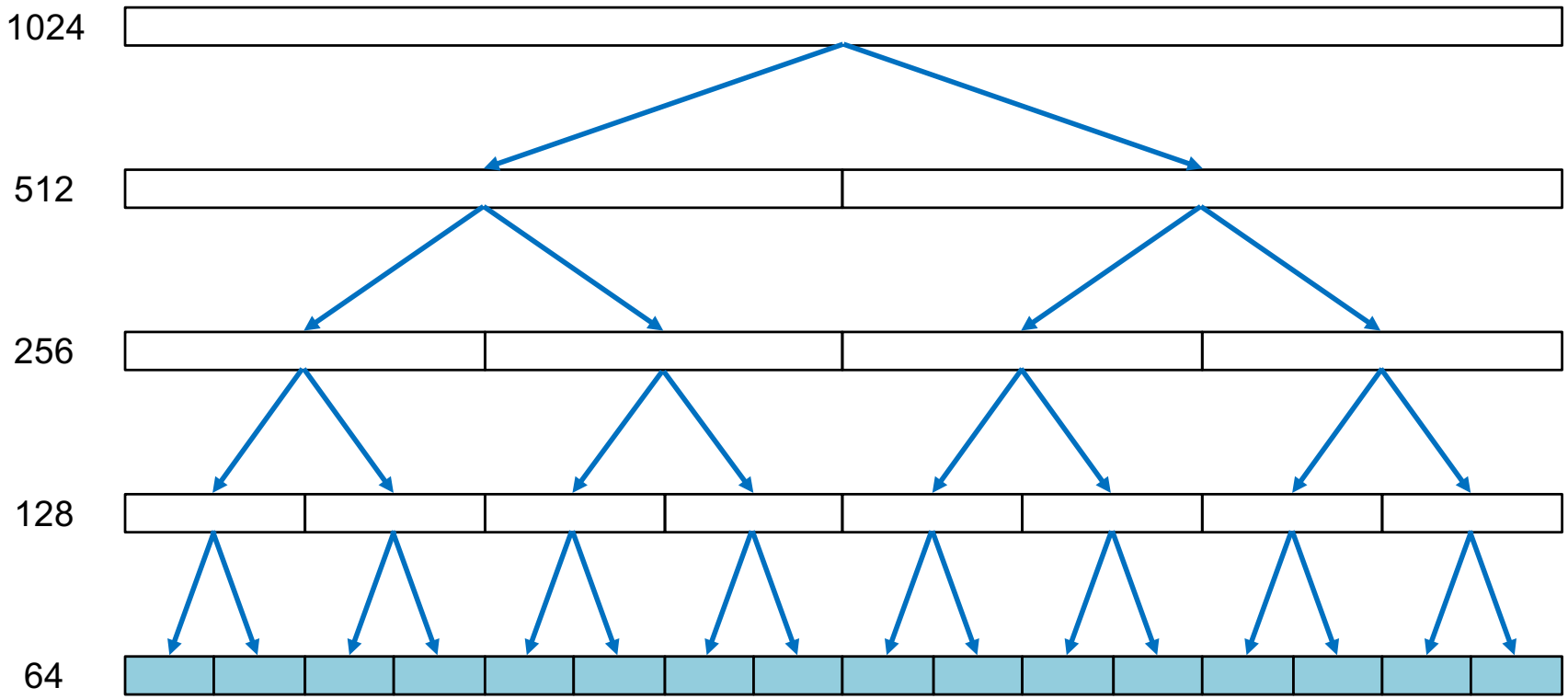
Example 4: "Divide–and–conquer" task decomposition

Sum of two vectors by recursively dividing 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);
   ...
}
```

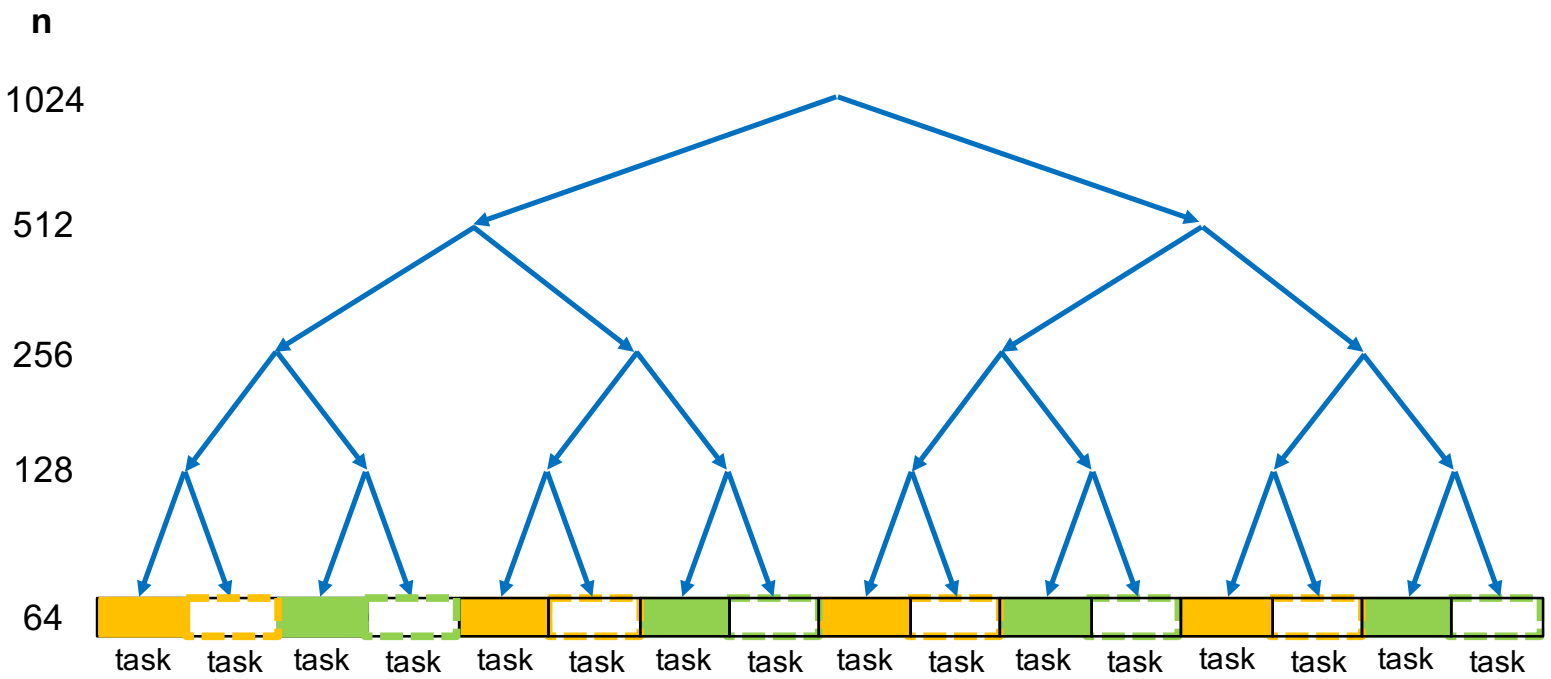
Example 4: "Divide–and–conquer" task decomposition

N=1024, MIN_SIZE=64



Two possible decomposition strategies

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



- Sequential generation of tasks

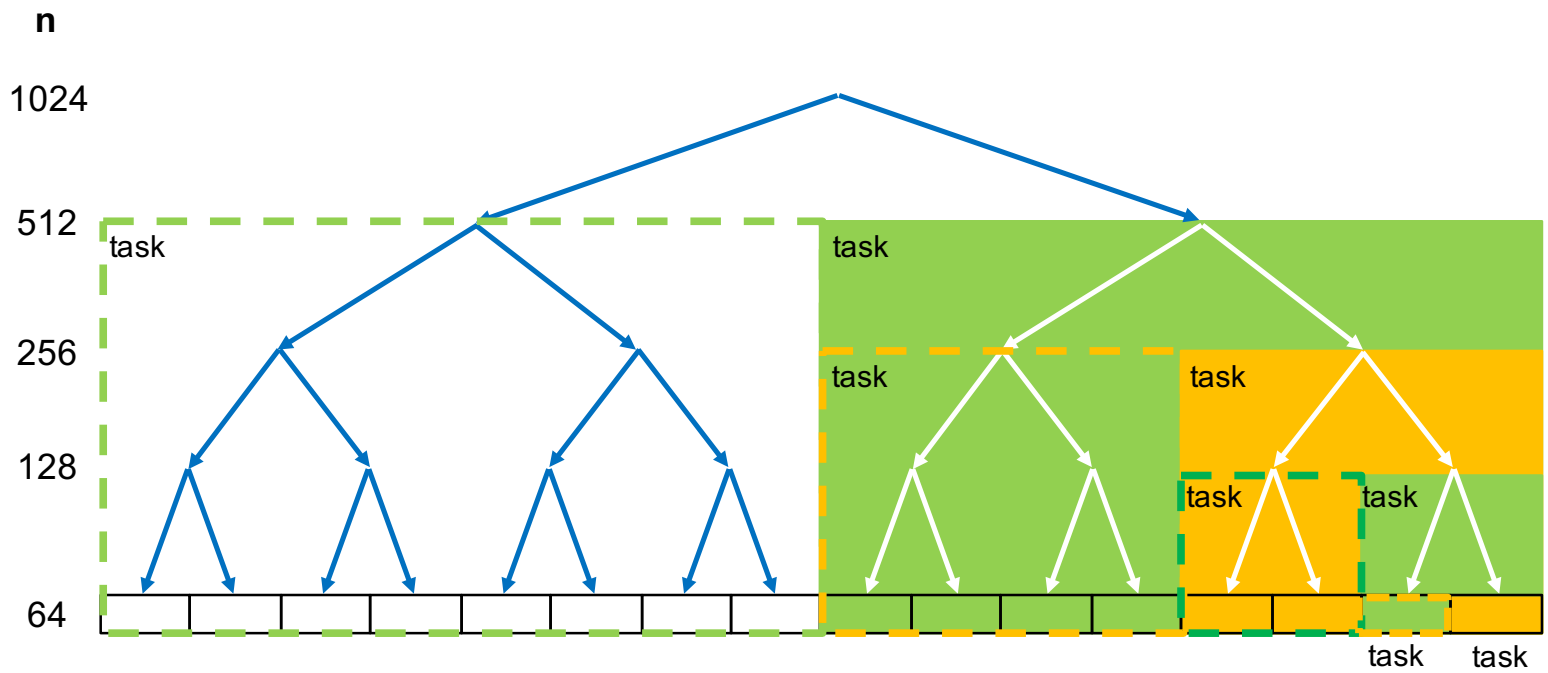
Implementing the decomposition strategies

Leaf parallelization

```
#define N 1024
#define MIN_SIZE 64

void 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
       #pragma omp task
       vector_add(A, B, C, n);
}

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

Two possible decomposition strategies (cont.)

- **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 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;
       #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);
   ...
}
```

Outline

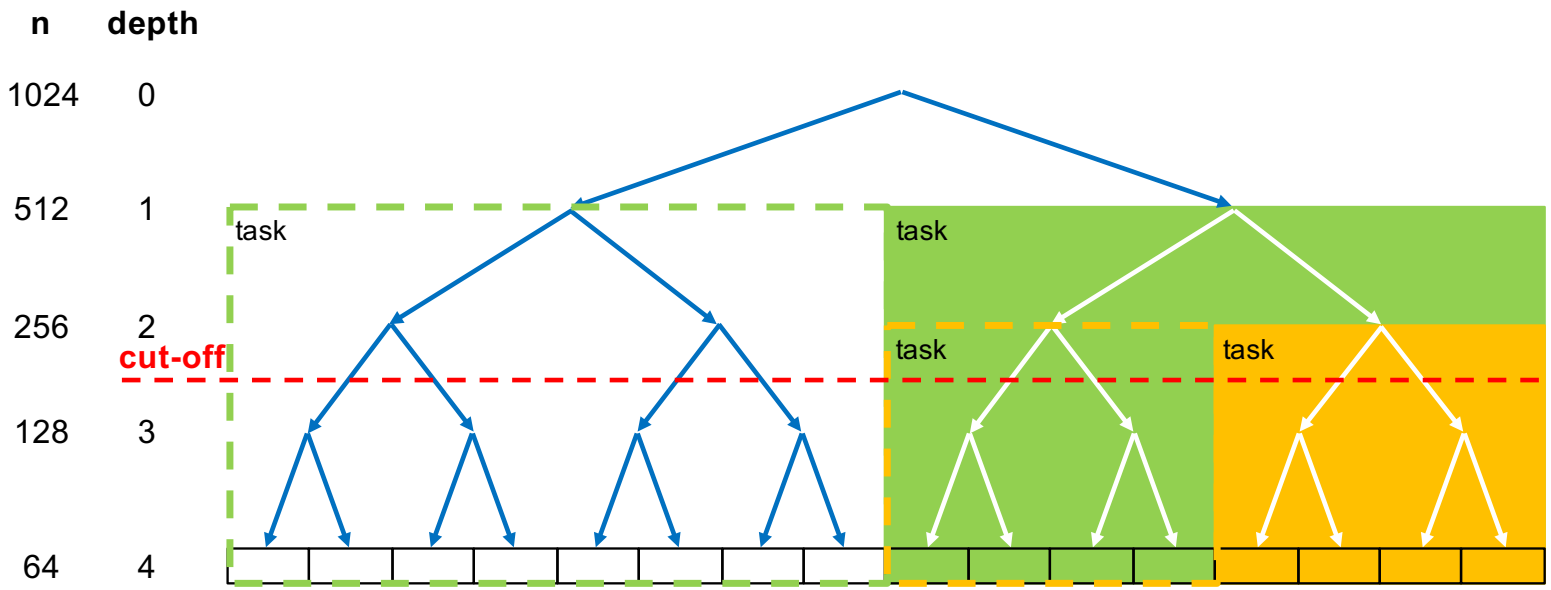
Task generation control

Excessive task generation if recursion depth is very high

- May cause excessive overhead
- It may be not necessary for certain problems
- Need 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**



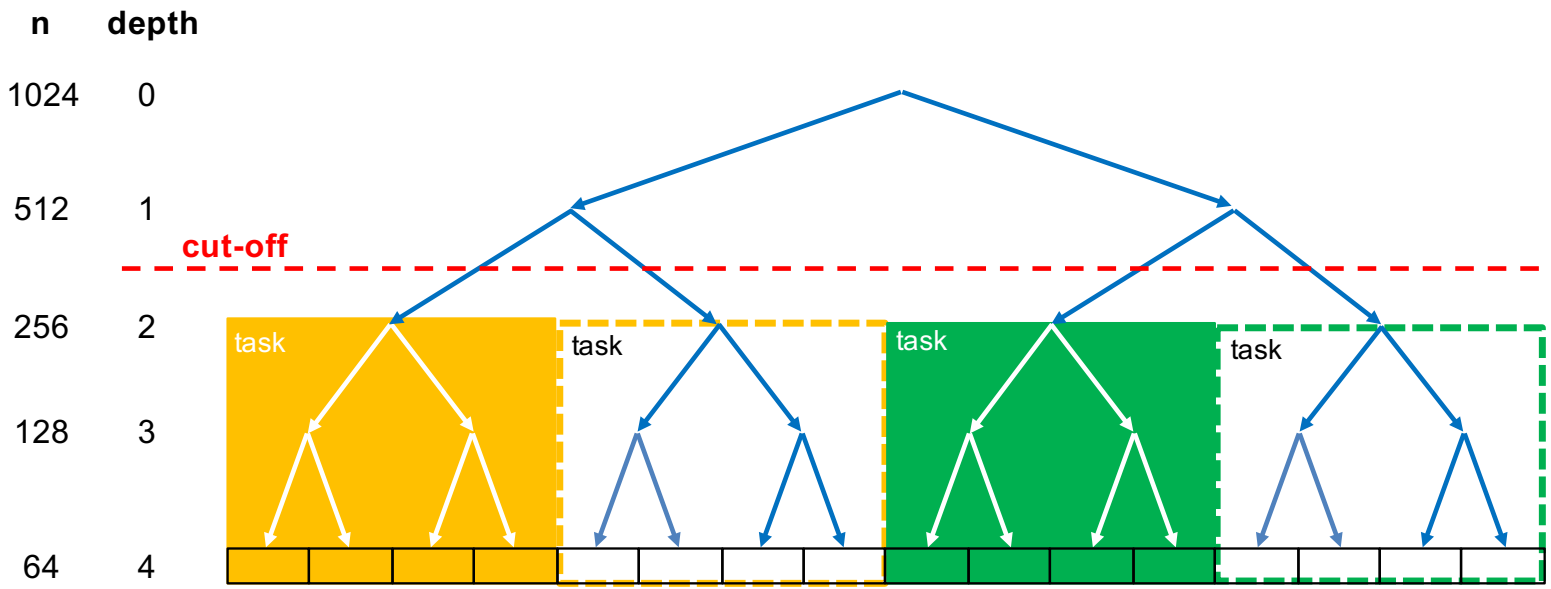
Cut-off control (cont.)

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);
}
...
```

Immediate task execution

- `if` clause: If the expression of an `if` clause evaluates to *false* the encountering parent task is suspended and the new one **undeferred** with respect it (but not necessarily executed by same thread). The parent resumes when the undeferred task finishes

- `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

Immediate task execution (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

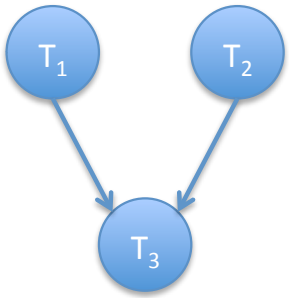
Very simple example with cut-off (rewritten)

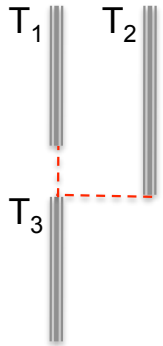
```
#define N 1024
#define MIN_SIZE 64
#define CUTOFF 3

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

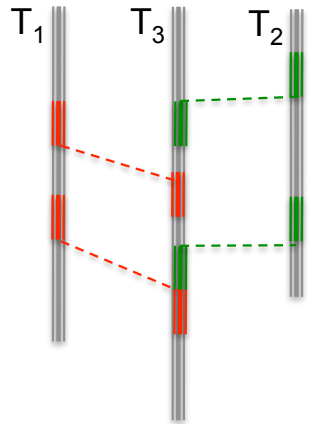
Outline

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

Task ordering 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 some data

- Task ordering constraints are easily imposed by sequentially composing tasks and/or using global synchronizations.

Task synchronization in OpenMP (summary)

- Thread barriers (`#pragma omp barrier`): wait for all threads to finish previous work
- 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
```

Fibonnaci series example

Sequential code

```
long fib(long n)
{
    if (n < 2) return n;
    return(fib(n-1) + fib(n-2));
}
void main (int argc, char *argv[])
{
      n = atoi(argv[1]);
      res = fib(n);
      printf("Fibonacci for %d is %d", n, res);
}
```

- The invocations of `fib` for `n-1` and `n-2` can be a task
- We need to guarantee that both instances of `fib` finish before returning the result

Fibonnaci series example (cont.)

OpenMP code using `task` and `taskwait`, with cut-off

```
long fib_parallel(long n, int d)
{
    long x, y;
    if (n < 2) return n;
    if (d<CUTOFF) {
        #pragma omp task shared(x) // firstprivate(n) by default
        x = fib_parallel(n-1, d+1);
        #pragma omp task shared(y) // firstprivate(n) by default
        y = fib_parallel(n-2, d+1);
        #pragma omp taskwait
    } else {
        x = fib_parallel(n-1, d);    // or fib(n-1)
        y = fib_parallel(n-2, d);    // or fib(n-2)
    }
    return (x+y);
}
void main (int argc, char *argv[])
{
      n = atoi(argv[1]);
      #pragma omp parallel
      #pragma omp single
      res = fib_parallel(n,0);
      printf("Fibonacci for %d is %d", n, res);
}
```

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.

Task dependences

- Example:
  - 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 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

- Example:
  - 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

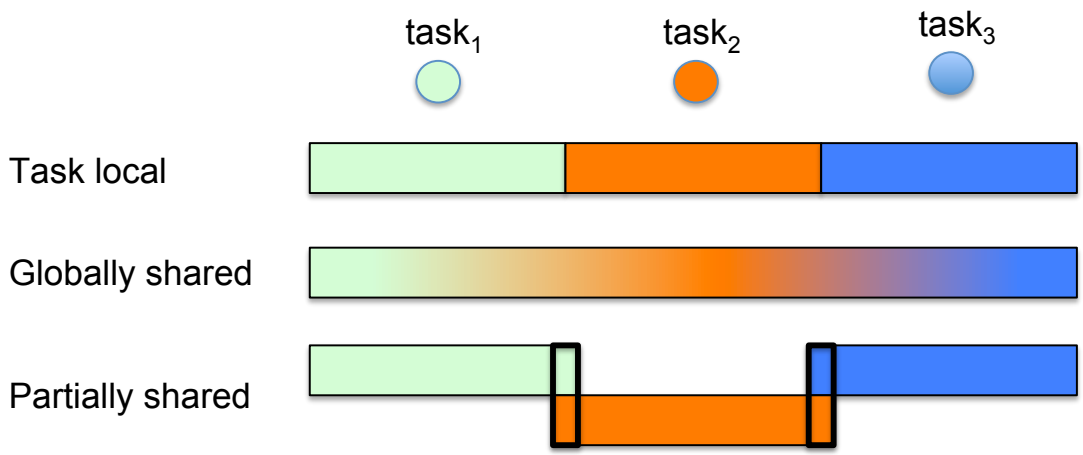
```
#pragma omp for collapse(2) ordered(2)
for (i = 1; i < N-1; i++) {
  for (j = 1; j < M-1; j++) {
    A[i][j] = foo(i, j);
    #pragma omp ordered depend(source)
    B[i][j] = alpha * A[i][j];
    #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
    C[i][j] = 0.2 * (A[i-1][j] + A[i+1][j]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

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

Outline

Data sharing constraints: patterns

- Task-local data: variables (or chunks of them) only used by single task
- Globally shared data: variables (or chunks of them) not associated with any particular task
- Partially shared data: variables (or chunks of them) shared among smaller groups of tasks (e.g. boundary elements or halos as in Gauss-Seidel)



Task interactions

- Task interactions are needed to guarantee proper access to shared data, without adding too much overhead
  - With shared memory architectures, all processors have access to all data, but must use synchronisation to prevent 'race conditions'
  - With distributed memory architectures, each processor has its own data, so race conditions are not possible, but must use communication to (in effect) share data (next chapter)
- Basic approach: first identify what data is shared, second figure out how it is accessed, and finally add the appropriate interactions to guarantee correctness

Protecting task interactions

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

- Two sequences of statements conflict if both access the same data and at least one of them modifies the data

Two mechanisms:

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

Task interactions: atomic construct

```
#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

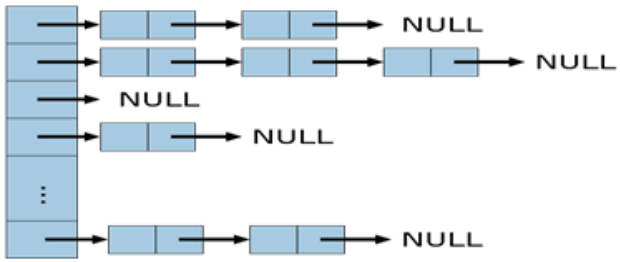
Task interactions: mutual exclusion

In OpenMP two alternatives for mutual exclusion: critical and low-level synchronization functions

- `critical (name)` pragma: if a thread arrives at the top of the critical-section block while another thread is processing the block, it waits until the prior thread exits
  - A critical section in OpenMP implies a memory `flush` on entry to and on exit from it (memory consistency)
  - `name` is an identifier that can be used to support disjoint sets of critical sections

Example: hash table

- Inserting elements in a hash table, defined as a collection of linked lists




```
for (i = 0; i < elements; i++) {
    index = hash(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(element[i]);
    #pragma omp critical
    insert_element (element[i], index);
}
```

Task interactions: mutual exclusion

- Low-level synchronization functions that provide a lock capability

```
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)
```

  - The lock functions guarantee that the lock variable itself is consistently updated between threads, ...
  - ... but do not imply a `flush` of other variables
  - Nested locks are possible → locking anomalies (deadlock)

Example: hash table

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

```
/* hash_lock declared as type omp_lock_t */
omp_lock_t hash_lock[HASH_TABLE_SIZE];

/* locks initialed in function main */
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(element[i]);
    omp_set_lock (&hash_lock[index]);
    insert_element (element[i], index);
    #pragma omp flush
    omp_unset_lock (&hash_lock[index]);
}

/* locks destroyed in function main */
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

- Group tasks that have strong data interactions. This also has an effect in increasing task granularity
- Replicate computations into local structures to avoid data sharing
- Separable dependencies (reductions): the dependencies between tasks can be managed by replicating key data structures and then accumulating results into these local structures. The tasks then execute according to the embarrassingly parallel pattern and the local replicated data structures are combined into the final global result

Parallelism (PAR)

Parallel programming principles: Task decomposition

Eduard Ayguadé and José Ramón Herrero

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2017/18 (Fall semester)