## Parallelism (PAR)

Mastering your task decomposition strategies: going some steps further

Eduard Ayguadé (Q2), José Ramón Herrero, Daniel Jiménez and Gladys Utrera

> Computer Architecture Department Universitat Politècnica de Catalunva

Course 2022/23 (Fall semester)

Reducing synch

### Additional learning material for this lesson

- Atenea: Unit 4 Task decomposition
  - Video lesson 6 (overview iterative vs. recursive task decompositions) and associated questionnaire
  - Additional Atenea quizzes
  - Going further: cut-off based on number of tasks pending to be executed (optional)
- ► Collection of Exercises: problems in Chapter 4

### Outline

#### Video lesson 6

Recursive task decomposition

### Concepts in video lesson 6

- 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 decomposition
  - Tasks = recursive procedure invocations, for example in divide-and-conquer problems
  - Examples: Fibonacci, multisort in lab session, graph exploration problems, ...



Reducing synch

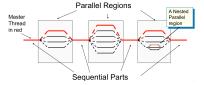
### Outline

Task creation and synchronization (Labs summary)

Recursive task decomposition

## Task creation in OpenMP (Labs summary)

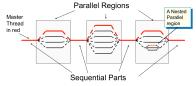
#pragma omp parallel: One implicit task is created for each thread in the team (and immediately executed).



- int omp\_get\_num\_threads: returns the number of threads in the current team. 1 if outside a parallel region
- int omp\_get\_thread\_num: returns the identifier of the thread in the current team that is executing a task, a value between 0 and omp\_get\_num\_threads()-1

### Task creation in OpenMP (Labs summary)

#pragma omp parallel: One implicit task is created for each thread in the team (and immediately executed). There is a barrier synchronization at the end of the parallel region.



- #pragma omp task: One explicit task is created, packaging code and data for (possible) deferred execution
- #pragma omp taskloop: Explicit tasks created for chunks of loop iterations. There is a taskgroup synchronization at the end of the taskloop.
  - In both cases, tasks executed by threads in the parallel region

## Task ordering in OpenMP (Labs summary)

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

```
#pragma omp task {}
                     // T1
                       // T2
#pragma omp task
   #pragma omp task {} // T3
#pragma omp task {}
                       // T4
#pragma omp taskwait
// Only T1, T2 and T4 are guaranteed to have finished at this point when T5 is created
#pragma omp task {}
                    // T5
#pragma omp task {}
                            // T1
#pragma omp taskgroup
                         // T2
    #pragma omp task
       #pragma omp task {} // T3
   #pragma omp task {}
                      // T4
// Only T2, T3 and T4 are guaranteed to have finished at this point when T5 is created
#pragma omp task {}
                          // T5
```

#### Outline

Task generation control Iterative task decompositions Recursive task decomposition

### Task generation control

Excessive task generation may not be necessary (i.e. cause excessive overhead): need mechanisms to control number of tasks and/or their granularity

- In iterative task decomposition strategies one can control task granularity by setting the number of iterations executed by each task
- In recursive task decomposition strategies one can control task granularity by controlling recursion levels where tasks are generated (cut-off control)
  - ▶ after certain number of recursive calls (static control)
  - when the size of the vector is too small (static control)
  - when there are sufficient tasks pending to be executed (dynamic control)
  - **.**..

### Outline

Video lesson 6

Task creation and synchronization (Labs summary)

Task generation control Iterative task decompositions

Recursive task decomposition

Reducing overheads and serialization due to synchronization

Hardware support for synchronization

Exploratory recursive problems

## Iterative task decomposition (1)

Task granularity defined by the number of iterations out of the loop each task executes. For example, using **implicit tasks**:

```
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);
```

Each implicit task executes a subset of iterations, based in the thread identifier executing the implicit task and the total number of implicit tasks (i.e., number of threads in the team).

## Iterative task decomposition (2) (optional)

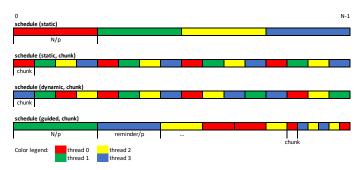
Using the work-sharing model in OpenMP (not covered in this course):

```
void vector_add(int *A. int *B. int *C. int n) {
   #pragma omp for schedule(static, BS)
   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 (optional)

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



## Iterative task decomposition (3)

Task granularity defined by the number of iterations each task executes. For example, using **explicit tasks**:

```
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):
```

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

## Iterative task decomposition (4)

#### Granularity: chunk of BS loop iterations

Option 1: requires loop transformation

```
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);
```

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

## Iterative task decomposition (5)

Option 2: taskloop construct to specify tasks out of loop iterations:

```
void vector add(int *A, int *B, int *C, int n) {
   int BS = \dots
  #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); ...
}
```

- **prainsize(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)

int main() {

### Iterative task decomposition: uncountable loop

List of elements, traversed using a while loop while not end of list

```
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:
```

Granularity is one iteration, hopefully with sufficient work to amortise task creation overhead.

Note: firstprivate needed to capture the value of p at task creation time to allow its deferred execution.

### Outline

#### Task generation control

Recursive task decomposition

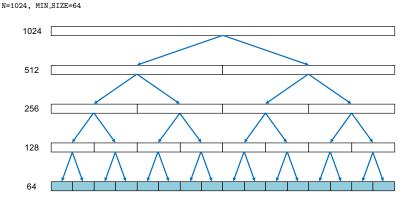
# Recursive task decomposition: divide—and—conquer (1)

#### Recursively divide the problem into smaller sub-problems

```
#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
       dot_product(A, B, n);
void main() {
   rec_dot_product(a, b, N);
```

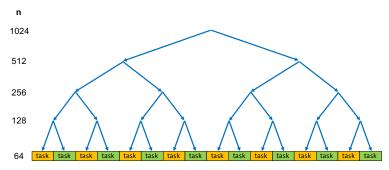
## Recursive task decomposition: divide—and—conquer (2)





## Recursive task decomposition: leaf strategy (1)

A task corresponds with each invocation of dot\_product once the recursive invocations stop



Sequential generation of tasks

# Recursive task decomposition: leaf strategy (2)

```
#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);
```

## Recursive task decomposition: leaf strategy (3)

```
#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;
      rec_dot_product(A, B, n2);
      rec_dot_product(A+n2, B+n2, n-n2);
   else
       #pragma omp task
      dot_product(A, B, n);
```

How could you reduce the overhead of updating variable result?

## Recursive task decomposition: leaf strategy (4)

```
#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: where is the task synchronization? (1)

```
#define N 1024
#define MIN SIZE 64
int result = 0:
void dot_product(int *A, int *B, int n);
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);
```

- Where is the task synchronization?
- Are there nested tasks?

# Leaf strategy: where is the task synchronization? (2)

```
#define N 1024
#define MIN_SIZE 64
int result = 0:
void dot_product(int *A, int *B, int n);
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);
   // Now we need the result here.
```

What kind of synchronization should we use? Where?

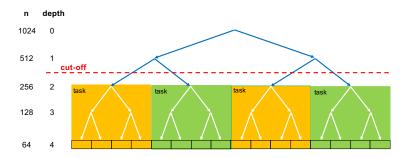


# Leaf strategy: where is the task synchronization? (3)

```
#define N 1024
#define MIN_SIZE 64
int result = 0:
void dot_product(int *A, int *B, int n);
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);
   #pragma omp takswait
   // Now we need the result here.
```

## How to control task granularity in leaf strategy (1)

#### Leaf parallelization with depth recursion control



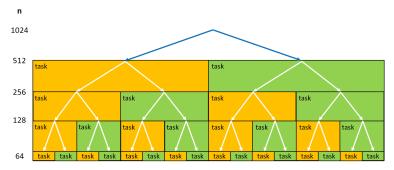
# How to control task granularity in leaf strategy (2)

### Leaf strategy with depth recursion 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);
    }
            // 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);
```

## Recursive task decomposition: tree strategy (1)

A task corresponds with each invocation of rec\_dot\_product



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

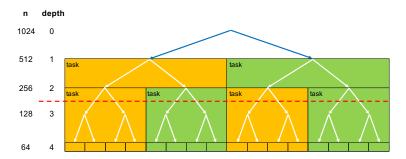
```
int dot_product(int *A, int *B, int n) {
   int tmp = 0;
   for (int i=0; i< n; i++) tmp += A[i] * B[i];
  return(tmp);
int rec_dot_product(int *A, int *B, int n) {
   int tmp1, tmp2 = 0;
   if (n>MIN_SIZE) {
       int n2 = n / 2;
      tmp1 = rec_dot_product(A, B, n2);
      tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
   } else tmp1 = dot_product(A, B, n);
  return(tmp1+tmp2);
void main() {
  result = rec_dot_product(a, b, N);
```

# Recursive task decomposition: tree strategy (2)

```
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);
```

### How to control task granularity in tree strategy (1)

#### Tree strategy with **depth recursion control**



# How to control task granularity in tree strategy (2)

### Tree strategy with **depth recursion 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 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.

## OpenMP support for cut-off: tree strategy

#### Making use of omp\_in\_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);
```

## Tree strategy: where is the task synchronization? (1)

```
Note: different tree strategy (using original seq code)... but not efficient!
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:
       #pragma omp task
       rec_dot_product(A, B, n2);
       #pragma omp task
       rec_dot_product(A+n2, B+n2, n-n2);
      // This taskwait is not needed: why?:
       #pragma omp taskwait
   } else dot_product(A, B, n);
void main() {
   #pragma omp parallel
   #pragma omp single
   rec_dot_product(a, b, N);
```

- Where is the task synchronization?
- Are there nested tasks?



## Tree strategy: where is the task synchronization? (2)

```
int result = 0:
void dot_product(int *A, int *B, int n);
void rec_dot_product(int *A, int *B, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       #pragma omp task
       rec_dot_product(A, B, n2);
       #pragma omp task
       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);
     // Now we need the result here.
```

What kind of synchronization should we use? Where?

# Tree strategy: where is the task synchronization? (3)

```
int result = 0:
void dot_product(int *A, int *B, int n):
void rec_dot_product(int *A, int *B, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       #pragma omp task
       rec_dot_product(A, B, n2);
       #pragma omp task
       rec_dot_product(A+n2, B+n2, n-n2);
   } else dot_product(A, B, n);
void main() {
   #pragma omp parallel
   #pragma omp single
     #pragma omp taskgroup
       rec_dot_product(a, b, N);
     // Now we need the result here..
```

Recursive task decomposition

Reducing overheads and serialization due to synchronization

## Avoiding task barriers: task dependences (1)

► The OpenMP runtime detects dependences between sibling tasks (i.e. from the same parent task) through the specification of the directionality for the variables used in the tasks

Task dependences are derived from the directionality type (in, out or inout) and its items in var\_list; this list may include array sections (e.g. v[0:n])

## Avoiding task barriers: task dependences (2)

- ▶ in specifier: 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 list <sup>1</sup>
- out and inout specifier: 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 list

#### Types of dependences:

- read—after—write: caused by matched out in
- write—after—read: caused by matched in out
- write—after—write: caused by matched out out

Note: if a list item is an array section, the matching should occur with an identically defined array section.

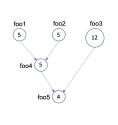
#### Example: wavefront execution with task dependences

- 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);
```

#### Serialisation caused by task barriers (1)

#### Given a TDG to implement with the OpenMP tasking model:



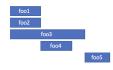


foo2()	op	oubii
1 0		taskwait
#pragma	omp	task
foo3()		
#pragma	omp	task
foo4()		
#pragma	omp	taskwait
#pragma	omp	task
foo5()		

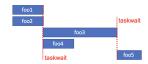
#pragma omp task

#nragma omn task

foo1()







foo5()

## Serialisation caused by task barriers (2)

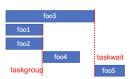
```
#pragma omp task
   #pragma omp task
   foo1()
   #pragma omp task
   foo2()
   #pragma omp taskwait
   #pragma omp task
   foo4()
   #pragma omp taskwait
#pragma omp task
foo3()
#pragma omp taskwait
#pragma omp task
foo5()
   foo1
    foo2
           taskwait
                    taskwait
             foo4
```

foo3

taskwait

foo5

```
#pragma omp task
foo3()
#pragma omp taskgroup
   #pragma omp task
  foo1()
  #pragma omp task
  foo2()
#pragma omp task
foo4()
#pragma omp taskwait
#pragma omp task
foo5()
```



#pragma omp task depend(out: a) foo1() #pragma omp task depend(out: b) foo2() #pragma omp task depend(out: c) foo3() #pragma omp task depend(in: a, b) depend(out: d) foo4() #pragma omp task depend(in: c, d)





## Additional functionalities (1) (optional)

taskwait with depend clause: instead of waiting for all child tasks to complete execution, it only waits for the predecessor child tasks according to the in, out and inout specifiers int x=0; y=2;

```
#pragma omp task depend(out: x) shared(x)
compute_short1(&x);
#pragma omp task shared(v)
compute_long(&v);
#pragma omp taskwait depend(in: x) // y not waited for at this point
printf("intermediate value for x=%d\n".x):
#pragma omp task shared(x)
compute short2(&x)
#pragma omp taskwait
printf("final values for x=%d; y=%d\n", x, y);
```

## Additional functionalities (2) (optional)

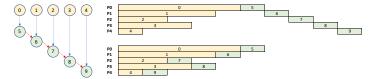
An iterator can be used in the depend clause, expanding to multiple values in the specifier they appear

```
for (i = 0; i < n; ++i)
    if (i%2) {
        #pragma omp task depend(out: v[i])
        compute element(&v[i], i):
#pragma omp task depend(iterator(it = 0:n), in: v[it])
                              // could also be depend(iterator(it = 1:n:2), in: v[it])
odd = sum odd elements(v. n):
even = sum even elements(v, n):
```

Note: this is not equivalent to the use of an array section in the in specifier (i.e. depend(in:v[0:n])), why not?

#### Additional functionalities (3) (optional)

mutexinoutset specifier: equivalent to inout but all dependent tasks can be executed in any order, one after the other



Red dependence expressed with depend(inout:x) (top temporal diagram) or with depend(mutexinoutset:x) (bottom temporal diagram). Observe that tasks can be executed in any order, but only one at a time.

#### Cancellation points in OpenMP (optional)

Tasks induced by exploratory decomposition can be terminated before finishing as soon as the desired solution is found

- #pragma omp cancel [parallel | taskgroup]: this directive activates the cancellation of the enclosing [parallel | taskgroup] region. The thread that finds the directive finishes its execution; the other threads continue their execution as normal.
- #pragma omp cancellation point [parallel | taskgroup]: introduces a point to check if cancellation has been activated. When found by a thread, if the enclosing [parallel | taskgroup] region has been already cancelled, then it finishes its execution.

# Cancellation points in OpenMP: very simple example (optional)

```
#pragma omp taskgroup
for (i=0; i<1000; i=i+100)
   #pragma omp task firstprivate(i) private(j)
        for (j=i; j<i+100; j++) {
            if (do_computation(j) == 0) {
                #pragma omp cancel taskgroup
            #pragma omp cancellation point taskgroup
```

The first task with 0 as a result of do\_computation will finalise the execution of all the tasks in the taskgroup

## Protecting task interactions in OpenMP (Labs summary)

#### Two mechanisms:

1. Atomic accesses: mechanism to guarantee atomicity in load/store instructions

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

- Atomic updates: x += 1, x = x foo(), x[index[i]]++
- Atomic reads: value = \*p
- Atomic writes: \*p = value

## Protecting task interactions in OpenMP (Labs summary)

#### Two mechanisms:

- 2. Mutual exclusion: mechanism to ensure that only one task at a time executes the code within a critical section
  - 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)
  - omp\_lock\_t OpenMP intrinsics and low-level synchronization primitives (next in this chapter)

## Reducing task interactions: overhead (1)

Reductions: replicate key data structures and locally working with these local structures; when appropriate, locally replicated data structures are combined into the final global result

```
int result = 0:
// Assume this function is instantiated as a task
void dot_product(int *A, int *B, int n) {
   for (int i=0: i< n: i++)
      #pragma omp atomic
      result += A[i] * B[i];
could be easily transformed into
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:
```

## Reducing task interactions: overhead (2)

Specifying reduction operations in explicit tasks generated with either task:

```
#pragma omp parallel
    #pragma omp single
       #pragma omp taskgroup task_reduction(+: sum)
        for (i=0; i < SIZE; i++)
            #pragma omp task firstprivate(i) in_reduction(+: sum)
            sum += X[i]:
or taskloop:
    #pragma omp parallel
    #pragma omp single
        // implicit taskgroup in taskloop construct
        #pragma omp taskloop reduction(+: sum)
        for (i=0: i < SIZE: i++)
            sum += X[i]:
```

#### Low-level synchronization functions using *locks*

**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)
```

#### Reducing task interactions: serialization (1)

Example: inserting elements in hash table defined as a collection of linked lists

```
HashTable
       element
```

```
typedef struct {
    int data:
    element *next;
} element;
int dataTable[SIZE TABLE]:
element * HashTable[SIZE_HASH];
for (i = 0: i < SIZE TABLE: i++) {
    int index = hash_function (dataTable[i], SIZE_HASH);
    insert_element (dataTable[i], index, HashTable);
```

## Reducing task interactions: serialization (2)

Easily parallelizable using an iterative task decomposition using taskloop. However ...

... updates to the list in any particular slot must be protected to prevent a race condition

```
typedef struct {
    int data;
    element *next:
} element:
int dataTable[SIZE TABLE]:
element * HashTable[SIZE HASH]:
#pragma omp taskloop
for (i = 0: i < elements: i++) {
    int index = hash_function (dataTable[i], SIZE_HASH);
   #pragma omp critical // atomic not possible here
    insert_element (dataTable[i], index, HashTable);
```

Serialization in the insertion of elements

## Reducing task interactions: serialization (3)

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

```
omp_lock_t hash_lock[SIZE_HASH];
#pragma omp parallel
#pragma omp single
for (i = 0; i < SIZE_HASH; i++) omp_init_lock(&hash_lock[i]);
#pragma omp taskloop
for (i = 0: i < SIZE TABLE: i++) {
    int index = hash function (dataTable[i], SIZE HASH);
   omp_set_lock (&hash_lock[index]);
    insert_element (dataTable[i], index, HashTable);
   omp_unset_lock (&hash_lock[index]);
for (i = 0; i < SIZE_HASH; 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

#### Outline

Video lesson 6

Task creation and synchronization (Labs summary)

lask generation control lterative task decompos

Recursive task decomposition

Reducing overheads and serialization due to synchronization

Hardware support for synchronization

Exploratory recursive problems

#### How are synchronizations implemented?

- In fact, entry to and exit from critical is the same as omp\_set\_lock and omp\_unset\_lock, respectively, but using an implicit (hidden) omp\_lock\_t variable
- atomic could also be implemented with omp\_lock\_t, but usually there is much better support at the architecture level (see later ...)
- ▶ How to implement lock-based synchronisation mechanisms?

#### Example: a simple, but incorrect, lock

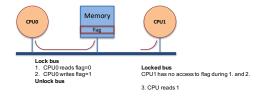
What's wrong with ...? (assume flag=0 means lock is free; taken otherwise)

```
CPUO
                                                 CPU1
           // omp_lock_t flag
                                                 // omp_lock_t flag
init_lock: st flag, #0
                                     init_lock: st flag, 0
set_lock: ld r1, flag
                                     set_lock: ld r1, flag
           bnez r1, set_lock
                                                 bnez r1, set_lock
           st flag, #1
                                                 st flag, #1
           // safe access
                                                 // safe access
unset lock: st flag, #0
                                     unset lock: st flag. #0
```

Problem: data race because sequence load-test-store is not atomic!

#### Support for synchronization at the architecture level

► Need hardware support to guarantee atomic (indivisible) instruction to fetch and update memory



test-and-set: read value in location and set to 1 Example: test-and-set based lock implementation

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

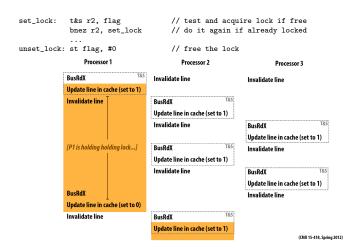
#### Support for synchronization at the architecture level

Atomic exchange: interchange of a value in a register with a value in memory

Example: atomic exchange based lock implementation

fetch-and-op: read value in location and replace with result after simple arithmetic operation (usually add, increment, sub or decrement). Valid to implement #pragma omp atomic

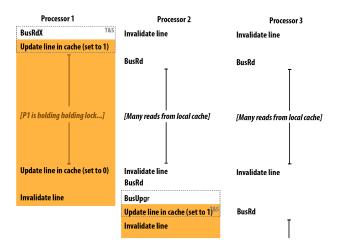
#### test-and-set lock coherence traffic



#### Reducing synchronization cost: test-test-and-set

- test-test-and-set technique reduces the necessary memory bandwidth and coherence protocol operations required by a pure test-and-set based synchronization:
  - Wait using a regular load instruction (lock will be cached)
  - ▶ When lock is released, try to acquire using test-and-set

#### test-test-and-set lock coherence traffic



#### Support for synchronization at the architecture level

- Atomicity difficult or inefficient in large systems. Alternative:
   Load-linked Store-conditional 11-sc
  - ▶ 11 returns the current value of a memory location
  - sc stores a new value in that memory location if no updates have occurred to it since the 11; otherwise, the store fails
  - sc returns success (1) or failure (0)
- Examples implementing atomic exchange (left) and fetch-and-increment (right):

```
// 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: 11 r2, location
   add r3, r2, #1
   sc r3, location
   beqz r3, try
```

#### Reducing synchronization cost: test-test-and-set

- test-test-and-set technique can also be implemented with 11-sc
  - First, wait using load linked instruction 11 (lock will be cached)
  - Second, use store conditional sc operation to test if someone else did it first

#### Other synchronization primitives

- How to implement a barrier synchronization primitive?
  - Threads arriving wait until all have reached the barrier
  - Structure with fields {lock, counter, flag}

▶ Does it work when consecutive barriers appear? Try to solve it

#### Outline

Video lesson 6

Task creation and synchronization (Labs summary)

Task generation control
Iterative task decompositions
Recursive task decomposition

Reducing overheads and serialization due to synchronization

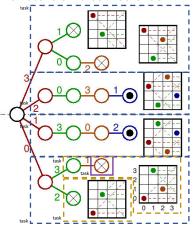
Hardware support for synchronization

Exploratory recursive problems

# How would you address the N-queens problem? (1)

#### How would you address the N-queens problem? (2)

For a 4x4 board, the recursion tree would be ...



## How would you address the N-queens problem? (3)

```
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>
        for ( int i=0 ; i < n ; i++ ) {
           a[i] = (char) i:
           if (ok(i + 1, a))
              #pragma omp task
                                                        // all firstprivate by default
              nqueens(n, j + 1, a);
     // Do we need to insert a task barrier at this point?
int main() {
     a = alloca(size * sizeof(char));
     #pragma omp parallel
     #pragma omp single
     nqueens(size, 0, a);
```

Do we need a new board for each task to be able to explore its own path? Is the implicit firstprivate(a) enough?

#### How would you address the N-queens problem? (4)

A new board has to be allocated if the path is explored as a task

```
void nqueens(int n, int i, char *a) {
     if (i == n)
          #pragma omp atomic
          sol count += 1:
     else {
        // try each possible position for queen <j>
        for ( int i=0 ; i < n ; i++ ) {
           a[i] = (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
                                                      // all firstprivate by default
              nqueens(n, j + 1, b);
        #pragma omp taskwait
```

Important: firstprivate(b) (implicit for new board) captures
the pointer to b, not the whole vector b

#### Where to dynamically allocate this memory?

- ptr=malloc(size): allocates memory block of given size
   (in bytes) in the heap, not initialized
- ptr=alloca(size): as malloc but within the current function's stack frame; this memory will be automatically deallocated from the stack when the current function returns!

**Important:** we must insert taskwait if using alloca. For malloc not strictly necessary, but we have to deallocate memory

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

```
void nqueens(int n, int j, char *a) {
     if (i == 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, i + 1, b):
            #pragma omp taskwait
        } else
           for ( int i=0 : i < n : i++ ) {
              a[i] = (char) i;
              if (ok(j + 1, a)) nqueens(n, j + 1, a);
           }
}
```

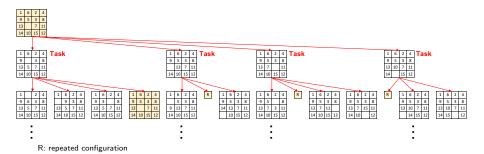
## Another example: 15-puzzle (without code) ... (optional)

The solution to a 15-puzzle (a tile puzzle). Possible movements of the empty cell: UP, RIGHT, LEFT and DOWN. Here we show a series of moves that transform a given initial state to the desired final state:

1	6	2	4	4		Ī	1	6	2	4		1	6	2	4		1	6	2	4		1	6	2	4		1	6	2	4
9	5	3	8	8		Ţ	9	5	3	8		9	5	3	8		9	5	3	8			5	3	8		5		3	8
13		7	11	1		Ī	13	10	7	11	$\overline{}$	13	10	7	11	_		10	7	11	$\overline{}$	9	10	7	11	_	9	10	7	11
14	10	15	12	2		Ī	14		15	12			14	15	12		13	14	15	12		13	14	15	12		13	14	15	12
		_																												
1		2	4	4	ſ		1	2		4		1	2	3	4	]	1	2	3	4	Ī	1	2	3	4	Ī	1	2	3	4
1 5	6	2	4 8	-	1	-	1 5	2	3	4		1 5	2	3	4 8		1 5	2	3	4		1 5	2	3	4		1	2	3	4 8
1 5 9	6	3	-	8		-	1 5 9	2 6 10	3			1 5	2 6 10	3	_		1 5 9	2 6 10	_	4 8 11		1 5					1 5	Ë	7	

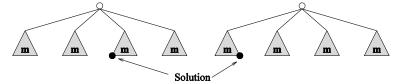
## Another example: 15-puzzle (without code) ... (optional)

The state space can be explored by generating various successor states of the current state and to view them as independent tasks



## Another example: 15-puzzle (without code) ... (optional)

Anomalous speed–ups of the parallel formulation of the problem: the speed–up depends on where the solution is found ...



- ▶ Left:  $T_1 = 2 \times m + 1$  and  $T_4 = 1$ , therefore ...  $S_4 = 2 \times m + 1$
- ▶ Right:  $T_1 = m$  and  $T_4 = m$ , therefore ...  $S_4 = 1$

And the parallel efficiency (i.e. how well used are processors)? Observe that on the right three processors waste their computation

## Parallelism (PAR)

Mastering your task decomposition strategies: going some steps further

Eduard Ayguadé (Q2), José Ramón Herrero, Daniel Jiménez and Gladys Utrera

> Computer Architecture Department Universitat Politècnica de Catalunva

Course 2022/23 (Fall semester)

Reducing synch