



## 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



## 4/82

## Outline

## Video lesson 6

## Task creation and synchronization (Labs summary)

## Task generation control

## Iterative task decompositions

## Recursive task decomposition

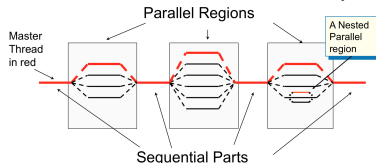
## Hardware support for synchronization

## Exploratory recursive problems



## 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 ...)









# 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):

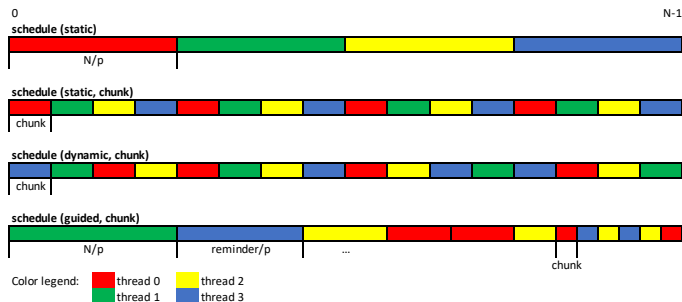
```
void vector_add(int *A, int *B, int *C, int n) {
    #pragma omp for schedule(static, BS)
    for (int i=0; i< n; i++)
        C[i] = A[i] + B[i];
}

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 = ...
    #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  $min(m, n)$

## Iterative task decomposition: uncountable loop

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

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

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

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

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

## 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;

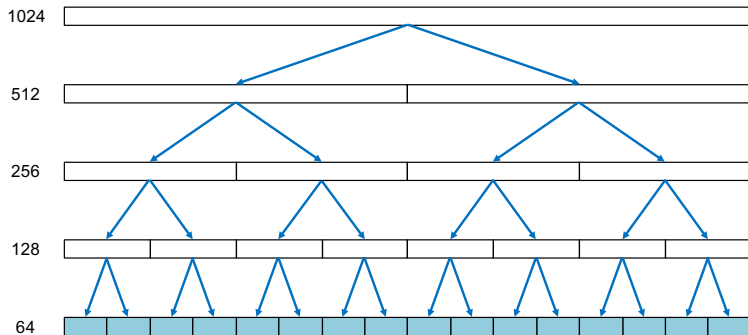
void dot_product(int *A, int *B, int n) {
    for (int i=0; i<n; i++)
        result += A[i] * B[i];
}

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);
}
```

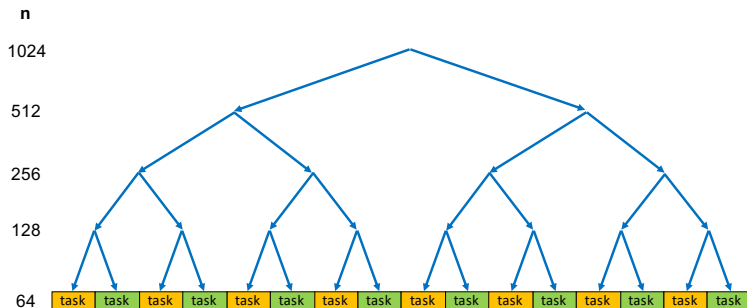
## Recursive task decomposition: divide-and-conquer (2)

N=1024, MIN\_SIZE=64



## 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;

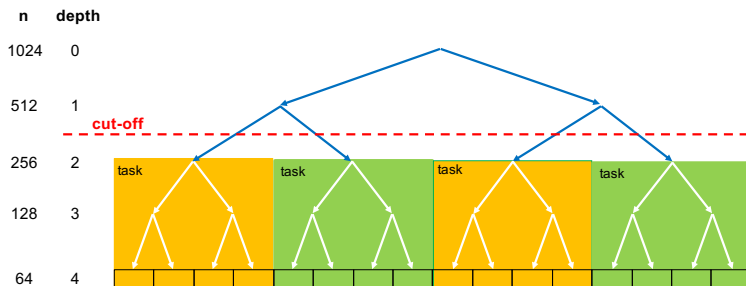
void dot_product(int *A, int *B, int n);

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 taskwait
        // Now we need the result here.
        ....
    }
}
```

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

## Leaf parallelization with **depth recursion control**



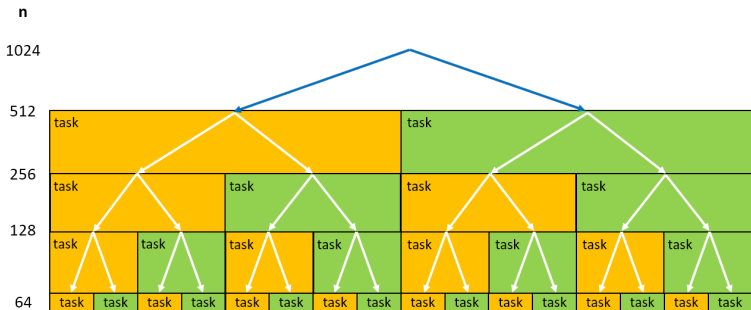
# How to control task granularity in leaf strategy (2)

## 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);
        }
    }
    else // if recursion finished, need to check if task has been generated
        if (depth <= CUTOFF)
            #pragma omp task
            dot_product(A, B, n);
        else
            dot_product(A, B, n);
}
...
```

# 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



# Recursive task decomposition: different sequential code ...

```
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);
}

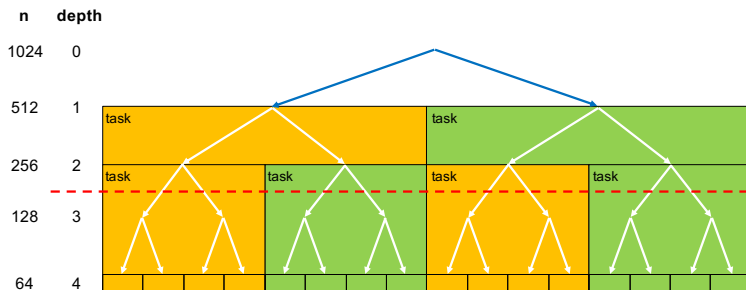
int rec_dot_product(int *A, int *B, int n) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        #pragma omp task shared(tmp1) // firstprivate(A, B, n, n2) by default
        tmp1 = rec_dot_product(A, B, n2);
        #pragma omp task shared(tmp2) // firstprivate(A, B, n, n2)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

```
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..
        ....
    }
}

```



## 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

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

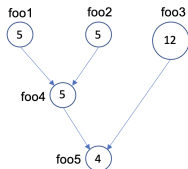
Task dependences are derived from the directionality type (`in`, `out` or `inout`) and its items in `var_list`; this list may include array sections (e.g. `v[0:n]`)





# Serialisation caused by task barriers (1)

Given a TDG to implement with the OpenMP tasking model:

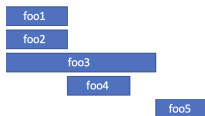


```

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

```

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





- ▶ `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

```
#pragma omp taskwait
printf("final values for x=%d ; y=%d\n", x, y);
```









## 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 intrinsics:

```
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)
```



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

- ```
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);
}
```

- 59/82



## Outline

## Video lesson 6

## Task generation control

## Iterative task decompositions

## Recursive task decomposition

## Hardware support for synchronization

## Exploratory recursive problems



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

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

## Parallelism (PAR)



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

## Example: atomic exchange based lock implementation

```

        mov r2, #1
set_lock:  exch r2, flag          // atomic exchange
          bnez r2, set_lock      // already locked?
          ...
unset_lock: st flag, #0          // free lock

```

- ▶ **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`

```

set_lock:   t&s r2, flag           // test and acquire lock if free
            bnez r2, set_lock      // do it again if already locked
            ...
unset_lock: st flag, #0           // free the lock

```











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





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

```

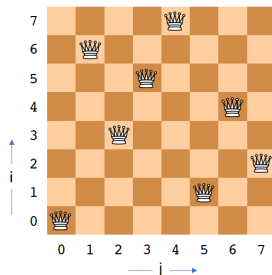
char *a; // Solution being explored
int sol_count = 0; // Total number of solutions found
int size = 8; // board size

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

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

```

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





## 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[j] = (char) i;
            if (ok(j + 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?

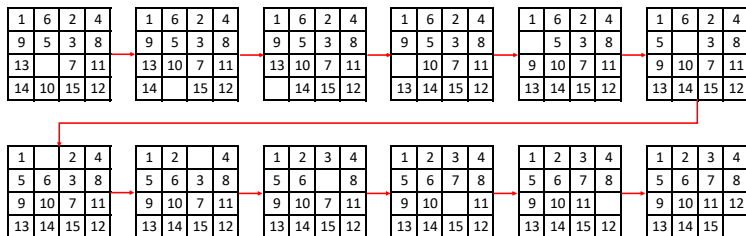




```
void nqueens(int n, int j, char *a) {
```

## 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:

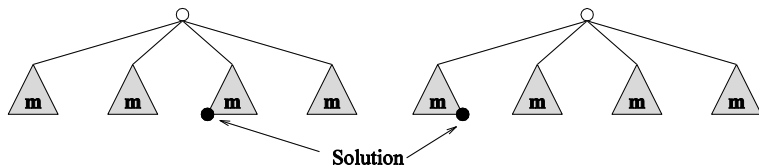






## Another example: 15-puzzle (without code) ... (optional)

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

