Task Decomposition

All construct used for task creation, be it implicit or explicit, accept the following **data-sharing** clauses:

- `shared` (default): the variable is the **same** inside and outside the construct. OpenMP doesn't put any restrictions to prevent a data race condition.
- `private` : the variable inside the construct is a **new variable** of the same type and **undefinde value**.
- `firstprivate` : the variable inside the construct is a **new variable** of the same type and is **initialized to the original value**.

Implicit tasks

Manual distribution

Each task executes a subset of iterations determined by the thread identifier and total number of threads.

Example:

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

`for` work-sharing construct

Each implicit task executes chuncks of itertations, depending on what is specified in the `schedule` clause.

`schedule` options:

- `static[, N]` : the iteration space is split into chunks of size `num_iterations`/`num_threads`, if a chunk size `N` is specified then that size is used.
- `dynamic[, N]` : iterations dynamically assigned in chuncks of size `N`. If `N` isn't specified then `N=1`.
- `guided[, N]` : similar to `dynamic` but chunk sizes decrease up to a minimum size `N`. If `N` isn't specified then `N=1`.

Other clauses:

- `collapse(N)` : allows to distribute work from `N` nested loops.

Example:

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

Explicit tasks: task generation and execution

`task` construct

Creates a new explicit task, packaging code and data.

Usually used with the `single` construct so as to not to repeat work.

Example:

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

`taskloop` construct

Specifies that the iterations of one or more associated loops will be executed in parallel.

Available caluses:

- `grainsize(N)` : all tasks will have size `N`.
- `num_tasks(N)` : there will be `N` tasks in total, each of size `num_iterations`/`N`.
- `collapse(N)` : allows to distribute work from `N` nested loops.
- `nogroup` : overrides the implicit associated `taskgroup`.

Example:

```
void vector_add(int *A, int *B, int *C, int n) {
 #pragma omp taskloop grainsize(5)
  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);
  ...
}
```

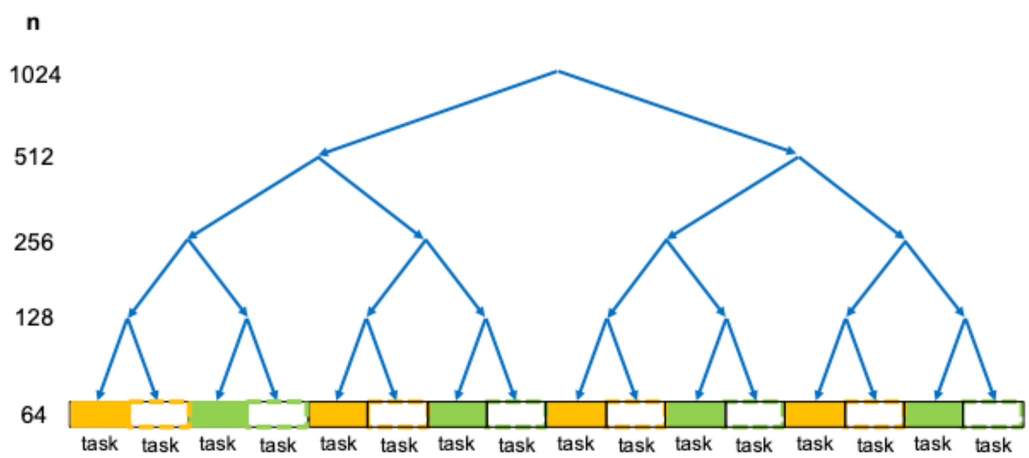
Recursive strategies

To understand the two approaches to a recursive implementation,we will use the following code as an example:

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

Leaf strategy

A **task** is created for each **leaf** in the recursion tree. Task generation is **sequential**.

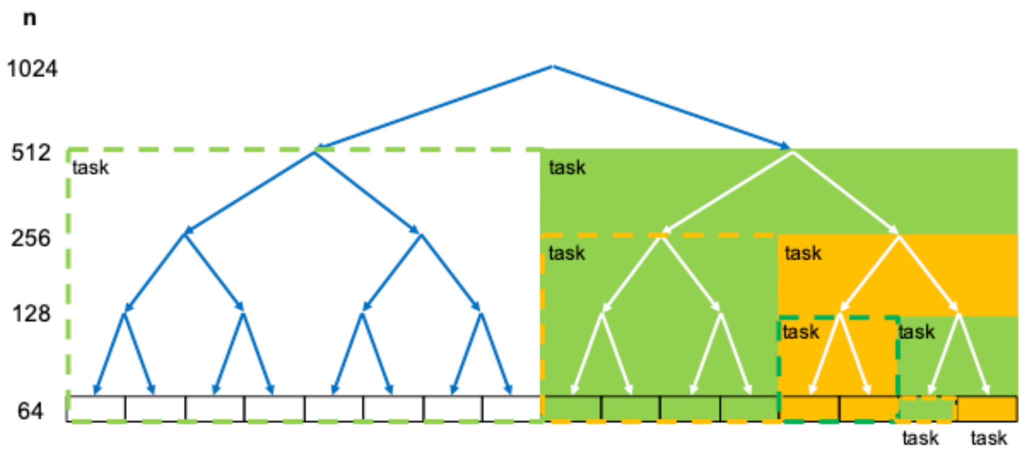


Code for leaf parallelization:

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

Tree strategy

A **task** is created for each **node** in the recursion tree. The tree traversal and task instantiation is **parallel**, with the overheads that it implies.



Code for 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);
  ...
}
```

Explicit tasks: task generation control

Cut-off control is necessary in recursive task decompositions:

- after a certain number of recursive calls
- when the number of generated tasks is too large
- ...

There are two OpenMP clauses for task creation constructs that help us with cut-off control:

- `final(cond)` : if `cond` evaluates to true, the generated task will be `final` and all its descendant tasks will be included in the generating task.
- `mergeable` : if the generated task is an `included` task, then the compiler ay choose to generate a `merged` task instead. A `merged` task's behaviour is as though there was no task directive at all.

Following the example of a tree strategy, below we can see how we might use the above clauses to implement cut-off control.

```
#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 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);
  ...
```

```
}
```

Data sharing

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

Two mechanisms:

- **Atomic**: guarantees atomicity in load/store instructions.
- **Mutual exclusion:** mechanism to ensure only one task executes the code at a time.

`atomic` construct

Ensures that a specific storage location is accessed atomically, avoiding simultaneous reading and writing. **Only protects** read/write operations.

More efficient than `critical`.

Example:

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

`critical` construct

A thread waits at the beginning of a critical region until **no other thread** is executing a critical region.

We can also use `critical(label)`, which allows to differentiate disjoint sets of critical sections.

Example:

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

Locks

Special variables that live in memory with two basic operations:

- **Acquire:** while a thread has the lock, the thread can do its work in private.
- **Release:** allow other threads to acquire the lock.

Methods:

- `omp_init_lock`: initialize the lock.

- `omp_set_lock` : acquires the lock.
- `omp_unset_lock` : releases the lock.
- `omp_test_lock` : tries to acquire the lock (won't block).
- `omp_destroy_lock` : frees lock resources.

Example:

```
omp_lock_t hash_lock[HASH_TABLE_SIZE];

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

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

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

`reduction(op:var)`

A task replicates `var` to **locally** work with it, when appropriate local copies are **combined** using the `op` operator.

It can be used as a clause for `parallel` and `for` constructs.

Example:

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

Task ordering contraints

`taskwait` construct

Suspends the execution of the current tasks and waits for the completion of its child tasks. It's a standalone directive.

```
#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 {}
```

`taskgroup` construct

Suspends the execution of the current task at the end of the structured block and waits for the completion of child tasks and their descendants.

```
#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
#pragma omp task {}
```

`depend` clause

OpenMP allows the specification of dependences between sibling tasks (from the same parent).

To declare the dependence we use the `depend(in/out/inout : var_list)` clause with the `task` construct.

Example:

```
#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);
    }
  }
}
```

Doacross loop

A loop nest where **cross-iteration dependences** exist.

Clauses:

- `ordered(N)` : defines the number of loops within the doacross loop.
- `depend` :
    - `depend(sink : expr)` : defines wait point for the completion of computation in a previous iteration defined by `expr`.
    - `depend(source)` : indicates completion of computation of current iteration.

Example:

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